12

SEP 05





RESUMEN DE MADRISX 2005

BASIC

ZXMINES, UN JUEGO COMENTADO

el aventurero

AVENTURAS MULTIPLATAFORMAS EN EL +3

ensamblador

INTRODUCCIÓN Y CONCEPTOS BÁSICOS

z88dk

SPRITES

hardware

RESEÑA DEL ADAPTADOR IDE A COMPACT FLASH DE JOSÉ LEANDRO

patas arriba

VEGA SOLARIS

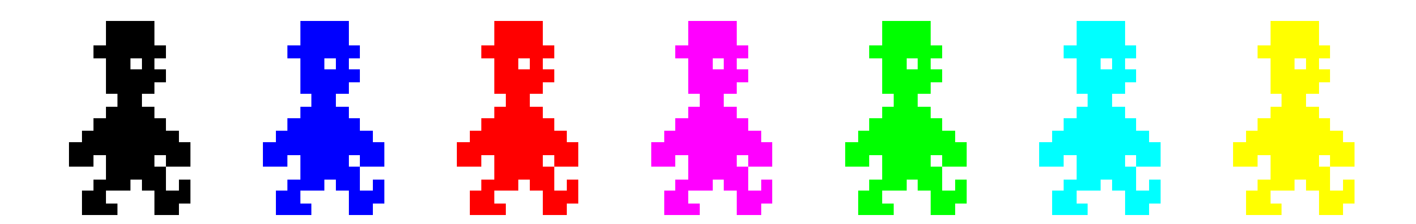
input

ENTREVISTA A QUASAR (AUTORES DE VEGA SOLARIS)

opinión

DISTRIBUCIÓN DENEGADA

http://magazinezx.speccy.org



No se puede decir que parece que fue ayer cuando publicamos el último número de la revista. De hecho, han pasado 7 larguísimos meses desde entonces, en los que no han dejado de suceder cosas en el mundo del Spectrum y durante los que algunos lectores ya nos echaban de menos.

El último número ya llegó con algo de retraso, pero la producción de éste poco a poco se fue torciendo, y al final ha sido una cuestión de orgullo personal y de, por qué no decirlo, cierto compromiso con la audiencia, lo que nos ha empujado y nos ha ayudado a finalizarlo.

Podemos decir que éste no creemos que sea el último número de Magazine ZX, pero sí que a partir de ahora haremos las cosas de diferente manera. Básicamente, abandonamos la publicación periódica. En adelante se publicarán sucesivas ediciones cuando estén preparadas, sin marcarnos plazos ni fechas. También hacemos un llamamiento a cualquiera que esté dispuesto a ayudar. Esta es una publicación abierta a vuestras colaboraciones.

Y ya sin más, pasamos a enumerar, como de costumbre, el contenido que nos atañe en esta ocasión, mucho más interesantes que las vicisitudes de esta redacción.

La sección de panorama incluye un amplio resumen de lo que fue la (ya lejana) reunión MadriSX 2005. A continuación, analizamos dos juegos, Frank'n'Stein, un clásico de 1984, y Moggy, recién salido de la factoría de CEZGames en este 2005.

Como habréis podido comprobar por la portada, gran parte del contenido de esta edición se centra en Vega Solaris, el juego que fue 'rescatado' en la MadriSX 2005. A él va dedicada la portada. También lo destripamos en nuestra sección Al Descubierto y HORACE nos ofrece una entrevista con sus creadores en INPUT.

Por otra parte, la ración de artículos técnicos también viene completita: destripamos ZXMINES, un juego creado en BASIC, SIEW nos enseña a mover sprites en Z88DK y SROMERO (NoP) comienza a impartir un interesante curso de Ensamblador para Spectrum.

En la parte de hardware, FALVAREZ presenta una de las creaciones del mago de la electrónica que es José Leandro Novellón: su lector de Compact Flash para Spectrum. Por otro lado, UTO nos introduce en el interesante mundo de las aventuras conversacionales multiplataforma, ejecutadas en un +3.

Para terminar, MIGUEL nos habla de su opinión acerca de la denegación de la distribución de títulos clásicos.

Esperamos que os guste el contenido que os ofrecemos y nos vemos en el próximo número.

Redacción de MAGAZINE ZX

editorial



Año III nº12 Sep 05

Redacción

Miguel A. García Prada
DEVIL_NET

Pablo Suau
SIEW

Federico Álvarez
FALVAREZ

Ilustración de Portada

Juanje Gómez
DEV

Colaboraciones en este número

Santiago Romero
NOP

Josetxu Malanda
HORACE

Carlos Sánchez
UTO

Maquetación en PDF

Javier Vispe
ZYLOJ

Contacto
magazinezx@gmail.com

índice

panorama

En esta entrega dedicamos nuestra sección de actualidad al evento quizás más importante del año en cuanto a reuniones de usuarios de ordenadores antiguos, la MadriSX & Retro.



El pasado 5 de marzo de 2005, más de 300 visitantes y 21 expositores se daban cita en el Centro Cultural El Greco de Madrid para formar lo que iba a ser la feria "MadriSX & Retro 2005": una reunión de usuarios de diferentes sistemas de 8 y 16 bits (dejando de lado PCs y otras consolas más potentes).

Tenemos que decir que el Centro Cultural se quedó pequeño para la gran cantidad de expositores y asistentes a la party: a lo largo de las aproximadamente 8 horas que duró (de 10:00 a 18:30) hubo algún momento en que la cantidad de público asistente era realmente espectacular para la difusión que puede tener una feria de este tipo. En momentos puntuales del día era realmente complicado desplazarse entre la marea de gente que paseaba entre los diferentes puestos de compra-venta o que participaba en las competiciones que se daban en algunos stands.



Aglomeración de asistentes cerca de la zona Spectrum

Entre los asistentes a MadriSX 2005 podemos destacar:

- Speccy.org
- MagazineZX
- El equipo de SPA2
- Ceinsur
- Sinclair QL: Recursos en Castellano
- Julio Medina y sus montajes de hardware
- VadeRetro
- Grupos de usuarios de Dreamcast (MadriDC), XBOX y otras consolas

Una de las cosas más atractivas para el visitante de MadriSX & Retro 2005 era la posibilidad de comprar gran cantidad de material de casi cualquier sistema retro a una precios más que asequibles: desde juegos de todo tipo de máquinas (ordenadores y consolas), hasta las máquinas en sí mismas, pasando por todo tipo de periféricos para ellas.

CONCURSOS Y VÍDEOS

Durante toda la jornada, los organizadores del evento iban amenizando el ambiente con diversos vídeos, concursos y rifas. Para ello se apoyaron en el sistema de sonido, y sobre todo, en una pantalla gigante situada al fondo de la sala, sobre la cabeza de los asistentes.

El tema de los vídeos era de lo más variopinto. Se mostraron cosas como parodias de videojuegos famosos, grabados con actores reales. Fue hilarante ver como una persona embutida en un disfraz de pieza de Tetris



Uno de los stands de compra-venta

intentaba encajar con todo el mobiliario urbano que se ponía a su alcance, o como un esqueleto le quitaba la ropa a un caballero andante que quedaba en cueros menores, al más puro estilo Ghost'n'Goblins. También hubo parodia de Metal Gear Solid, en el que fue el vídeo más largo de todos. El público también pudo disfrutar de partidas grabadas en las que se mostraba como se batían juegos como el Castlevania o el Super Mario Bros. 3 en un tiempo récord; francamente impresionante.

En lo referente a concursos y a participación del público, hubo tres tipos de actos, básicamente. En varios momentos del día se hizo una rifa en la que, ayudados por un programa de MSX que generaba números aleatorios (o al menos eso decían ellos), los organizadores premiaban con diferentes productos vintage a aquellas personas cuyo número de entrada se correspondiera con el que se cantaba en ese momento. Varios de los expositores se animaron a donar material para que pudiera ser rifado de esta manera.

Algo que generó gran expectación, o al menos hizo que la gente callara durante unos

minutos, fue el concurso de músicas de sistemas vintage. Durante un breve periodo de tiempo sonaba la música de un juego cualquiera de una plataforma cualquiera por los altavoces, y el cometido del concurso era adivinar de que juego se trataba, obteniendo más puntos si además se acertaba a que sistema pertenecía dicha versión del juego. Para participar se tenía que levantar la mano y esperar a que viniera una persona con el micro para que pudieras dar la respuesta. Evidentemente, con la cantidad de gente presente, este formato de participación se mostró ciertamente ineficiente, beneficiando a aquellas personas que estaban más cercanas precisamente al organizador que portaba al micrófono. De todas maneras, se percibía un subidón friki en el ambiente cuando alguien era capaz de adivinar no solo que esa música era la del Bubble Bobble (cosa relativamente sencilla), sino que además pertenecía a la música original de la recreativa... sin embargo, hubo otros títulos no tan fáciles de adivinar. Por último, ya por la tarde, se organizaron partidas a un par de juegos Matra, que podían ser seguidas también en la pantalla gigante.



Sin duda, ponerle nick a todas estas caras no tiene precio

ZONA SPECTRUM

Pero sin duda, lo más destacable a nivel humano fue la reunión de usuarios de Spectrum: el hecho de asociar caras a los nicks y nombres que manejamos habitualmente en el IRC y las news. Y es que estábamos prácticamente casi todos. Quizás desde los stands no relacionados con Sinclair el punto de vista era distinto, pero para los que se encontraban en esa parte del recinto, estaba claro que la inmensa mayoría del movimiento se concentró en esa zona. Siempre se podía ver un grupo de gente conversando cerca de los stands spectrumneros. No solo eso, sino que la gente además aprovechaba para intercambiar cintas, revistas y Cds.



Varias personas observaban atónitas como unas personas ya mayorcitas se lo pasaban pipa con un juego de fútbol de hace 15 años

Quizá, la actividad que más llamó la atención a aquellos que se acercaron al stand de Speccy.org, y que tuvo lugar durante el desarrollo casi completo del día, fue la competición de International Match Day. Este juego de fútbol, que podría producir la risa a cualquiera de los aficionados a las últimas novedades en juegos de este tipo, demostró que los años le han tratado bien y que es capaz de generar tanta emoción, diversión, e incluso algunas veces crispación como los videojuegos actuales.

Los jugadores se encargaban de suplir la falta de realismo gráfico con una gran dosis de ingenio e imaginación, haciendo comentarios como: "¡Toma, que remate de cabeza!", "Mira como te lo he metido con un remate en plancha", e incluso con con radiocomentaristas aficionados que, ante situaciones como un pase raso a la banda no tardaban en comentarle a todo el mundo: "observad como ha hecho un pase interior y ahora se va por la banda... ¡que jugadón!".

En el caso de los asistentes había más división. Estaban desde los que miraban con cara de desconcierto a los participantes del torneo, preguntándose cómo podían estar divirtiéndose con un juego como ese, a los que también hacían el mismo tipo de comentarios: "Mira hijo, yo jugaba a eso... ¿has visto como le ha robado el balón limpiamente y como ha despejado?".

En el campeonato, que por cierto ganó Fede Alvarez después de derrotar a Z80user en la final, pudimos observar momentos para el recuerdo (y de gran tensión), como por ejemplo, el gol en el minuto 89 de NoP que le clasificaba para las semifinales y dejaba a Siew fuera, o el partido de la final. A veces parecía imposible reprimir lo que se sentía y los jugadores se levantaban de la silla chillando y celebrando los goles por todo lo alto.



El partido más polémico, entre Siew y NoP, que se saldó con un empate tras marcar este último en el minuto 89

Y mientras tanto... los chicos de SPA2 pasando cintas a TZX como locos y escaneando carátulas, y peleándose con el Vega Solaris para poder digitalizarlo. Desde luego, unos incansables trabajadores.

JUEGOS PARA SPECTRUM

Como diría cierto conocido personaje, para los aficionados al Spectrum fue motivo de orgullo y satisfacción comprobar como la gente no solo disfrutaba de la gente y de productos o juegos programados hace años, sino que incluso se presentaban cosas nuevas, nuevos juegos para nuestra plataforma.

Por su parte, el stand de CEZ GS (Computer Emuzone Games Studio), del cual formaba parte Beyker (al que todos aquellos habituales de ECSS conocerán), ponía a la venta tres nuevos juegos de Spectrum en formato físico (uno de los cuales, Pitfall ZX, ya se encontraba publicado previamente en Internet). Estos juegos eran, además del

nombrado Pitfall ZX, Galaxy Fighter y Run Bill Run.


Este era el aspecto profesional de los juegos que nos ofrecía Beyker

En este punto cabe destacar la excelente producción de estos nuevos juegos, que incluían cajas y carátulas profesionales, incluyendo también sus instrucciones. También mostraron un par de juegos para otras plataformas, en concreto un clon del Columns para CPC y un juego llamado Stratos para MSX, que no se encontraban a la venta.

Pero sin duda, lo que mas llamaba la atención a quienes pasaban por delante de ese stand eran... ¡unas cajas de berberechos y unos pequeños pollitos de colores! Sin duda, todo un guiño a lo que habían estado siendo los últimos acontecimientos en el grupo de news es.comp.sistemas.sinclair en las últimas semanas.



Pollitos y berberechos; quien no lo entienda, es que no se ha conectado a ECSS durante mucho tiempo

El grupo Compiler Software, como parte del stand Speccy.org, también presentó sus nuevas producciones en formato físico. Estos juegos no se encontraban a la venta, solo se mostraban como exposición, y pueden ser obtenidos a través de su página web. Los tres juegos eran los siguientes:

- ZX Poker: no es más que una conversión del típico juego de poker en solitario, en el que deberemos ligar jugadas y no quedarnos sin blanca.
- ZX Columns: conversión para Spectrum del clásico Columns de Sega, realizado utilizando la herramienta Z88DK, de la que se habla también en los últimos números de Magazine ZX.
- Another Brick on the Wall 2: continuación del juego machacaladrillos que obtuvo la victoria en una de las categorías del concurso de BASIC de Bytemaniacos del año 2004. Entre las novedades, la aparición de nuevos ladrillos o el acercamiento de la paleta a la pared opuesta si pasamos mucho tiempo en la misma fase, complicando nuestro objetivo.


Hubo una excepción. Loatlan SÍ que se pasó la primera fase del Another Brick on the Wall 2

Además de descargar los juegos desde la página de Compiler Software, también es posible obtener las carátulas, en el caso de que queramos imprimirlas y crear nuestras propias cintas. Como detalle curioso, durante todo el día estuvo Another Brick on the Wall 2 funcionando en uno de los Spectrums del stand, pero no se pudo ver a nadie capaz de pasar de la primera pantalla. ¿Se habrán pasado estos chicos con la dificultad?

Por lo tanto, el gozo nos invade cuando vemos que hay tanta gente interesada en el Spectrum como para llenar los stands durante todo el día e incluso presentar nuevos programas.

REMAKES

Por otro lado, y para celebrar el 20 aniversario de su creación, se presentó el remake de uno de los primeros juegos de Dinamic: Babaliba.

Enmarcado en el proyecto REDO, propuesto por la web remakeszone.com, el objetivo con este juego era realizar un clon para PC con sistemas operativos Linux y Windows, actualizando sus gráficos y su música pero manteniendo la jugabilidad y el espíritu de las primeras creaciones españolas para el Spectrum.

Este remake también puede ser descargado de la web de Compiler Software.

SPA2: PRESERVANDO A TODA MÁQUINA

Si de alguien nos tenemos que sentir orgullosos todos los aficionados al Spectrum es sin duda de los titanes que se hicieron cargo del stand de SPA2, Miguel y Juan Pablo, que pasaron el día entero trabajando preservando material original.


Mucho trabajo y nada de descanso

Desde primera hora de la mañana, se encargaron sin descanso de digitalizar cintas y portadas de la gente que las iba llevando. Jose Manuel se les unió más tarde, ayudando en el difícil cometido de tratar de pasar a TZX un juego que no llegó a ser publicado comercialmente...

A las 12 del mediodía, aproximadamente, estalló la bomba: Fernando y Carlos, invitados por Horace, pasaron por

el stand portando cintas y papeles con el material de su obra no publicada, Vega Solaris. Y desde esa hora, Jose Manuel se unió para que entre todos, al final del día, pudieran acabar con la difícil tarea de conseguir disponer de ese juegazo en versión TZX, así como todo el código, mapas, etc. escritos a mano en papel, escaneados. Realizar el TZX fue una labor muy ardua que duró varias horas, pero al final el esfuerzo valió la pena y ahora todos podemos disfrutar de esa pequeña obra de arte.

Debido a este gran acontecimiento, muchas cintas quedaron sin pasar a TZX, ya que el tema del Vega Solaris adquirió total prioridad. Sin embargo, mucha gente, como Horace, tbrazil, Javier Herrera, Metalbrain, z80User, Miguel, etc., dejaron sus cintas en manos de esta gente tan competente para que, con tranquilidad y buenos alimentos, pudieran terminar el trabajo en días sucesivos. Estas cintas serían devueltas más tarde a sus dueños. Alguno incluso llevó otro tipo de material, como Horace, que hizo llegar al Trastero diversos recortes del Pequeño País referidos al Spectrum.

Quien sin duda se portó muy generosamente también fue Radastan, que donó varias de las cintas del stand de Ceinsur a la gente de SPA2 y El Trastero, para que también pudieran preservarlas.

LA PARTY DEL HARDWARE

En lo referente al Spectrum, esta party ha tenido también su representación a nivel de hardware: por un lado pudimos ver en vivo y en directo el SuperCartucho de Jose Leandro, quien también nos mostró cómo utilizaba un disco duro de PC en el +3E de Fede Álvarez.

Por otro lado, la mesa de Julio Medina estaba repleta de hardware, con gran cantidad de montajes, entre ellos muchos de los que nos llevaba a casa la magnífica revista MicroHobby.

VEGA SOLARIS

Uno de los hechos más destacables de la MadriSX de este año ha sido sin duda la recuperación y presentación del juego Vega Solaris, programado por Fernando Sáenz Pérez y Carlos García Cordero.



Fernando Sáenz posando junto a Vega Solaris

A lo largo de los diferentes artículos de este número de MagazineZX hemos pretendido dar a Vega Solaris la cobertura informativa que se merece, como habréis podido comprobar.

Con respecto a lo que fue la recuperación del juego en sí mismo, Fernando y Carlos llegaron a la MadriSX & Retro 2005 llevando una cinta de audio que contenía lo que iba a ser el mayor hallazgo de software de Spectrum del año, y acompañando ésta con una carpeta de folios que documentaban el desarrollo del juego, incluyendo mapas de memoria, mapas de las localidades del juego y rutinas en ensamblador. El material aportado por Fernando y Carlos fue tan completo, que incluía microdrives, diferentes cintas (con rutinas, versiones beta, etc.) e incluso la documentación presentada en el Registro de Propiedad para registrar los derechos del juego.

Con bastante paciencia y después de algún que otro susto con la cinta (que inicialmente se resistió a ser convertida a TZX), el equipo de SPA2 junto a otros colaboradores del mundillo del Spectrum lograron obtener un fichero TZX para uso y disfrute de la comunidad de usuarios de Sinclair: fue el colofón para un día inolvidable para todos aquellos que tuvimos la suerte de asistir.

EN RESUMEN

MadriSX & Retro 2005 fue una fiesta para todos los usuarios de microordenadores de 8 bits que se dieron cita en la capital de España.

Podríamos decir que fue especialmente fructífera para los usuarios de sistemas Sinclair (no sólo Spectrum, también QL), por la gran cantidad de actividades, exposiciones de proyectos y nuevos juegos, y, sobre todo, por el ambiente y la posibilidad de disfrutar en vivo y en directo con todos aquellos compañeros con los que mantenemos contacto a distancia a través de Internet.

No todo fue positivo en MadriSX & Retro 2005: la iluminación de la sala era bastante pobre lo cual deslucía un poco la toma de fotografías y vídeos.

Además, el escaso tiempo de que dispusimos los asistentes (sólo 8 horas) fue a todas luces insuficiente para todas las ganas de diversión que habían en el ambiente: no se podía estar en todas las cosas a la vez (preservación de software, competiciones, conversaciones o intercambios y compra-ventas de material), de modo que muchos de nosotros nos quedamos con las ganas de conocer más proyectos, conversar en más profundidad con más asistentes o incluso despedirnos adecuadamente de todos los compañeros que conocimos allí.

También hubieramos agradecido algo más de espacio, ya que la ubicación de Speccy.org y MagazineZX acabó siendo una de las esquinas del Centro Cultural donde habían elementos de escenario que nos impedían movernos con la libertad que hubiéramos querido.

No obstante, estos pequeños detalles son insignificantes al lado de todo lo que ocurrió en MadriSX & Retro 2005, se pierden dentro del mar de lo que fue las presentaciones, los nuevos proyectos, las competiciones, la competitividad entre los jugones del Matchday,

los maratones de preservación de software o el mero hecho de estrechar la mano a personajes tan insignes dentro del mundo del Spectrum como los que se dieron cita allí.

Desde MagazineZX os invitamos a que forméis parte de todo esto en el año 2006, para que la representación del Spectrum en esta party sea (como este año) un referente a seguir y no una simple anécdota.

Pero no todo ha sido MadriSX desde que se publicó el número anterior, han pasado multitud de cosas más. Vamos a tratar de comentar las que nos han parecido más relevantes.

JUEGOS

Durante estos meses hemos podido contemplar el nacimiento de nuevas producciones para nuestros ordenadores, lo cual es la mejor noticia que podríamos dar, ya que indica que la plataforma no está muerta ni vive exclusivamente de la nostalgia. Los títulos más destacables son:

- Area 51
- Ghost Castles
- Run Bill Run
- Beastie Feastie
- Moggy

Además, Josep Coletas sigue incansable con su serie de juegos basada en las aventuras del Dr. Van Halen:

- Los cantos de Anubis
- Tristes alas del destino
- Último acto

Las podéis encontrar en la página web del Club de Aventuras AD.

SOFTWARE

En todo este período de tiempo se han lanzado nuevas versiones de algunos emuladores y utilidades que solemos emplear. La lista es afortunadamente extensa, pero citaremos los más relevantes:

- PSPectrum, el primero emulador de Spectrum para la Sony PSP.
- Pasmo, un buen ensamblador de Z80.
- BASin, entorno de desarrollo en BASIC.
- Aspectrum, el emulador portable de Spectrum.
- Z88DK, compilador de C para plataformas de 8 bits.
- DSP, otro emulador de Spectrum.

SPECCY.ORG

Como sabéis, gracias al patrocinio de Radastan, speccy.org ha cambiado de alojamiento, de forma que ahora se puede prestar en mejores condiciones y no se espera que se sufra la exasperante lentitud que lo estaba mermando últimamente.

Por otro lado, después de muchos meses "bajo mínimos", el portal speccy.org volvió a dar servicio a la comunidad de Spectrum hace apenas un mes, habiendo recuperado todas las noticias, usuarios y comentarios anteriores y volviendo a ser uno de los puntos de reunión de los aficionados. Sus mantenedores también han aprovechado para remozarlo estéticamente.

También hay que destacar que el movimiento en torno a las webs alojadas en el portal no cesa. Recientemente se ha producido una gran actualización de SPA2, en la que se ha añadido un montón de software publicado en nuestro país. Y también hay que dar la enhorabuena a Horace y su buscador MHoogle, que nos permite bucear de forma sencilla en los contenidos de la revista Microhobby.

PUBLICACIONES

En cuanto a las publicaciones dedicadas al Spectrum, una de cal y otra de arena. ZXF se despide de sus lectores con el número 10. En principio se trata de un "hasta pronto", pero en estos casos nunca se sabe. La publicación que sigue pasito a pasito asentándose es la española ZX Spectrum Files (ZXSF), que publicó su tercer número y cuyos creadores ya están preparando la cuarta entrega. También debe estar al caer la quinta entrega de First Generation.

REUNIONES Y CONFERENCIAS

El mundillo retro sigue generando eventos. Prueba de ello han sido las dos conferencias sobre 8 bits y retroinformática celebradas en este período de tiempo.

En abril se celebró en Zaragoza una mesa redonda en la que participaron ponentes de la talla de Gonzo Suárez, Enric Cervera, Fernando Sáenz y David López Guaita.

Unos meses después, en julio, durante la Euskal Party y en el marco de lo que se denomina Retro Euskal, tuvo lugar una interesante conferencia en la que participaron Marcos Jouron, Ricardo Puerto y Fernando Sáenz.

CONCURSOS

Hará unos meses que concluyó el Concurso Arcade 2005, con la única participación de Beyker, quien presentó una buena conversión de Beastie Feastie (y que ganó merecidamente una consola GP32 de premio, todo sea dicho). Sólo faltó algo más de participación para haber hecho el veredicto algo más animado.

Siguiendo la estela de años anteriores, recientemente se ha

puesto en marcha un nuevo concurso de BASIC patrocinado por Radastan. En esta ocasión parece ser que habrá 3 categorías, una de BASIC puro, otra de BASIC libre sin ensamblador (pero sí con la posibilidad de invocar a las rutinas de la ROM) y otra de BASIC con rutinas en ensamblador. La convocatoria oficial no se ha efectuado todavía en la página web, por lo que puede sufrir modificaciones. El plazo de entrega parece ser que concluirá en diciembre.

Afortunadamente, en el futuro se seguirán celebrando este tipo de concursos, lo que supone un continuo acicate a que los programadores presenten sus piezas.

REDACCIÓN DE MAGAZINE ZX

http://madrisx.cjb.net/ Madrisx

http://members.fortunecity.com/jonathan6/egghead/ Area 51

http://wss.sinclair.hu/ Ghost Castles

http://computeremuzone.com/cezgs/index.php?id=jcsilver Run Bill Run

http://www.redeya.com/bytemaniacos/arcade2005/beastie.zip Beastie Feastie

http://cezgs.computeremuzone.com/ficha.php?id=7 Moggy

http://caad.mine.nu/ CAAD

http://www.redeya.com/bytemaniacos/ Bytemaniacos

http://personal.telefonica.terra.es/web/exkq/pspectrum/ PSPectrum

http://www.speccy.org/spa2/spanish/whatsnew.htm SPA2

http://www.arrakis.es/~ninsesabe/pasmo/ Pasmo

http://sourceforge.net/projects/aspectrum/ Aspectrum

ftp://ftp.worldofspectrum.org/pub/sinclair/emulators/pc/windows/BASin_r12.exe BASin

http://z88dk.sourceforge.net/ Z88DK

http://leniad.cjb.net/index.htm DSP

http://www.microhobby.com/zxsf/pagina_1.htm ZXSF

http://www.speccy.org/ Speccy.org

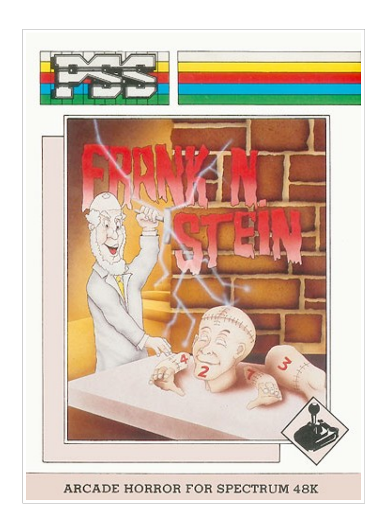


análisis

En este número presentamos el análisis de Frank'N'Stein de la mano de MIGUEL y de Moggy, una de las últimas novedades presentadas para Spectrum, por FALVAREZ.

FRANK'N'STEIN

Título	Frank'N'Stein
Género	Plataformas
Año	1984
Máquina	48K
Jugadores	1 Jugador
Compañía	PSS
Autor	Colin Stewart
Otros comentarios	Crash Sinclair User



Si hay un programa en la historia de los videojuegos, al menos en lo que al Spectrum se refiere y me atrevería a decir que en ámbito general también, que marcó una época, un estilo y abrió nuevos caminos ese fue, sin duda, Manic Miner. El juego creado por Smith inauguraba un nuevo género: los plataformas.



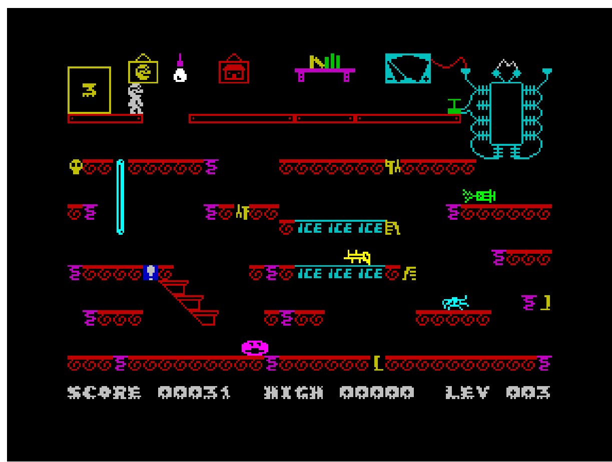
Menú de opciones

A raiz de ver la luz el minero Willy empezaron a salir clones, más o menos acertados, sobre el mismo estilo, como el que hoy nos ocupa. Aunque bien es cierto que Colin Stewart, autor del que parece el único juego para Spectrum en su carrera, lejos de copiar la fórmula o de intentarlo al menos, incluye alguna novedad que hacen de este Frank N Stein un juego altamente adictivo y un título de calidad.

El argumento está basado en uno de los libros de terror más famoso de todos los tiempos: Frankenstein. Redactado en el año 1818 por Mary Shelley, a la adolescente edad de 18 años.

A finales del siglo XIX, en un ténebre castillo en la frondosidad del bosque Negro, el doctor Frank N. Stein busca en su laboratorio la manera de dar vida a un monstruo. El cuerpo lo realiza con trozos de cadáveres y planea devolverlo a la vida con una fuerte descarga eléctrica. Y aquí tomamos el control manejando al científico en su arduo trabajo.

La mecánica del juego es muy sencilla y el viaje a través de las cincuenta pantallas que conforman el juego se nos hará ameno. En las pantallas impares debemos construir el cuerpo del monstruo, que está dividido en 7 partes y esparcido por el laboratorio. A continuación le aplicamos una descarga eléctrica, y habremos conseguido nuestro objetivo. Pero la victoria no es fácil, y nuestra creación despierta, como muchos de nosotros, de mala leche, y se revelerá contra su hacedor. A continuación en las pantallas pares tenemos que dormir o quitar la vida a nuestro amigo, para lo cual volveremos a aplicarle una nueva descarga eléctrica.



Una de las pantallas del juego

Para complicar las cosas nuestro laboratorio está repleto de obstáculos, seres y objetos que intentarán evitar que demos vida a nuestra locura. La pantalla está conformada en vista lateral y los siete trozos en los que está dividido el cuerpo, a saber, cabeza, dos partes del pecho, dos de las caderas y las dos piernas, están esparcidos por toda la habitación. Para complicarnos las cosas no podemos recoger las partes en el orden que nos venga bien, hay que hacerlo en un orden definido: primero la cabeza, luego el pecho derecho, el izquierdo a continuación y así hasta la pierna izquierda con lo que conseguiremos recomponerlo en su totalidad.
Los variados enemigos se mueven en una secuencia predeterminada, recorrido que tendremos que estudiar detenidamente para calcular el momento en el que les podemos esquivar sin riesgo para nuestra salud. Además, a lo largo de las diferentes fases encontraremos objetos que, o bien nos ayudarán en nuestra misión, o bien nos obstaculizarán. Placas de hielo que harán que nos movamos sin control en una cierta distancia, muelles para saltar a un nivel superior, ya que nuestro protagonista no es muy deportista y no sabe hacerlo, barras de bombero para descender de grandes alturas, unas bombillas que nos aturdirán por unos momentos y no dejarán que nos movamos pero, por contra, incrementarán el tiempo que nos resta para llevar a cabo nuestra misión de reconstrucción y un largo etcétera de items.

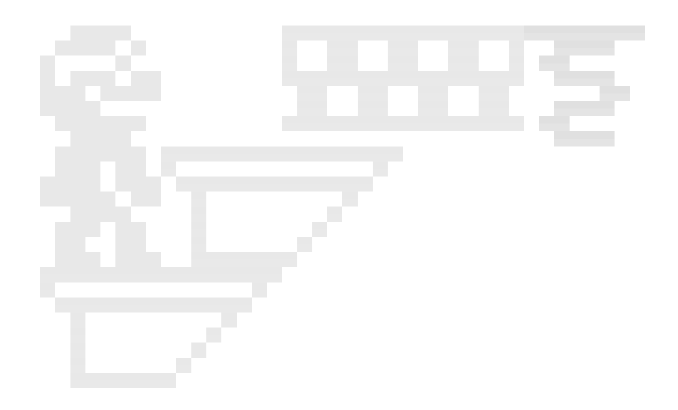
El tiempo, representado por una especie de voltímetro junto al monstruo que estamos creando es uno de nuestros grandes problemas. Corre rápido, con lo que no tenemos mucho tiempo para detenernos a pensar la forma de solucionar la pantalla, pero tenemos que hacerlo si queremos llevar a buen fin nuestra tarea. Contra más alto esté el voltímetro a la hora de aplicar corriente al monstruo, de peor humor se despertará y en la siguiente pantalla lo notaremos en su agresividad. Por lo menos nos quedan las citadas bombillas, que harán que decremente el nivel y con ello tenemos más tiempo para finalizar la fase.

En las pantallas pares la cosa cambia un poco. Tenemos que desconectar al monstruo con otra descarga, para lo que tenemos que llegar a la zona superior de la pantalla a través de varios pisos. Además el monstruo nos enviará unos regalitos en forma de bola con toda la intención de impactarnos.

Y pasando a analizar la parte técnica del juego, tengo que decir que los gráficos, teniendo en cuenta que son del año 1984, están muy bien realizados, tanto los sprites como los escenarios y objetos. Son coloridos y bien definidos. El sonido es sencillo, carece de una melodía tanto en el menú como a lo largo de la partida, pero por contra los diferentes efectos son agradables y no hacen daño al oído por exceso, aunque unicamente están representados los pasos y caídas de nuestro personaje.

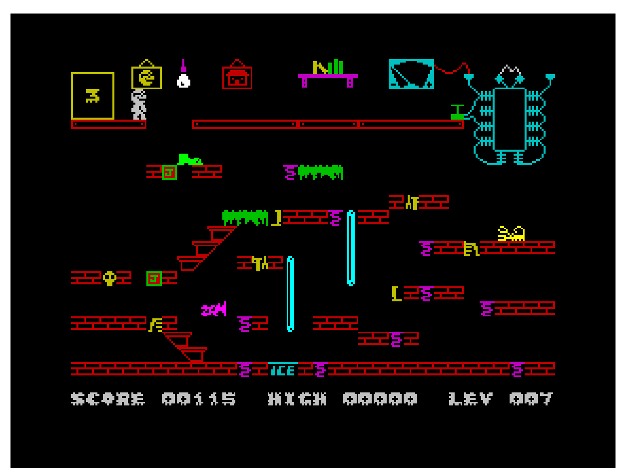
El movimiento del juego es preciso y Frank responde bien a nuestras pulsaciones. Podemos desplazar a nuestro personaje horizontalmente y dejarlo caer para perder altura, pero cuidado con esto: si dejamos caer al doctor desde muy alto perderemos una de las tres vidas de las que disponemos. Para ello, cuando las alturas sean excesivas, o bien deberemos buscar un camino alternativo o bien utilizar una "barra de bombero" si la hay disponible. Para ganar altura no hay más que un método, y es situarnos encima de un muelle y pulsar la tecla de acción, con ello iremos al nivel inmediatamente superior, pero cuidado con los enemigos... El juego se puede controlar con el teclado, utilizando únicamente tres teclas, o mediente joysticks Kempston, Protek o Sinclair.

Un único pero hay que poner al juego. En un par de ocasiones el programa ha devuelto el control al BASIC con errores, interrumpiendo la partida. Hay que decir que ha sido un par de veces aisladas, una sin motivo, con el mensaje "invalid colour" y otro error ha sido provocado al pulsar "break" aunque no sucede cada vez que se acciona esta tecla.



Resumiendo, esta creación de PSS es un juego muy ameno y entretenido, sin llevar al frenetismo de Manic Miner o Roller Coaster a la hora de calcular el salto al píxel, pero teniendo que administrar bien el tiempo y calcular muy bien las

rutas de los diferentes enemigos. Es un juego que te hace pensar, no se basa únicamente en la acción, y es algo de agradecer.



Vamos avanzando y la cosa se complica

trucos

Puedes encontrarlos en The Tip Shop

descárgalo de

WOS

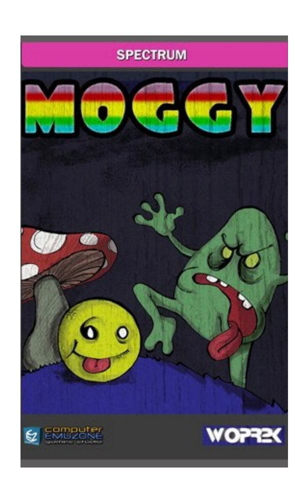
Valoraciones

originalidad	[7]	jugabilidad	[9]
gráficos	[7]	adicción	[10]
sonido	[5]	dificultad	[8]

MIGUEL

MOGGY

Título	Moggy
Género	Arcade
Año	2005
Máquina	48K/128K
Jugadores	1 Jugador
Compañía	CEZGS
Autor	na_th_an, Anjuel, Beyker
Otros comentarios	N/A





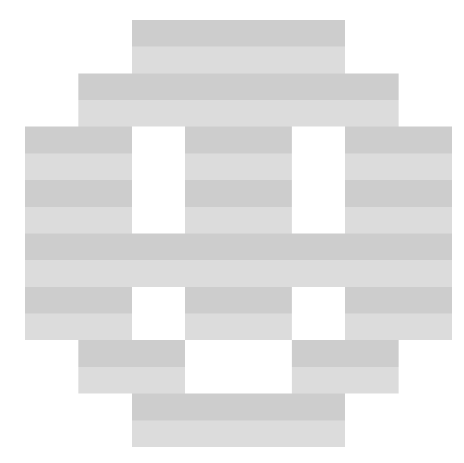
Menú principal

Lamentablemente, en pocas ocasiones tenemos la oportunidad de comentar un juego prácticamente "recién salido del horno". Ésta es una de ellas. Moggy fue publicado hace unas semanas por los chicos de Computer EmuZone Games Studio, que están haciendo un gran trabajo a la hora de apoyar nuevos desarrollos para las plataformas de 8 bits.

Moggy es un arcade de concepción sencilla, basado en una idea muy simple, pero que nos hará pasar muy buenos ratos. El manual nos habla de un hechicero que ha drogado a los bolotes con la intención de usar su energía para conseguir más poder. Pero ahí está Moggy, el alma de la fiesta, que en estado de embriaguez tratará de rescatar a sus compañeros.

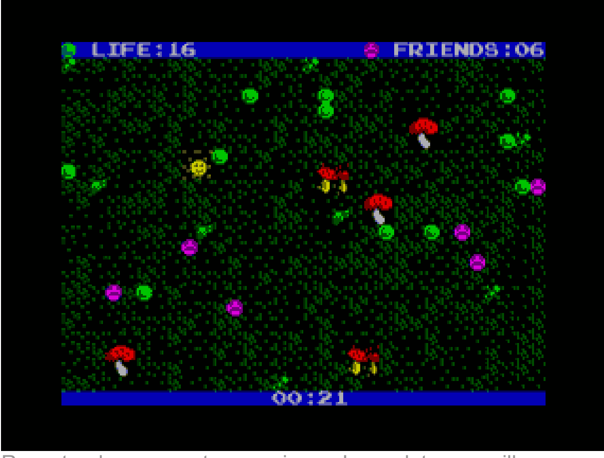
El juego nos presenta 25 pantallas en las cuales tendremos que liberar a nuestros amigos

simplemente tocándolos. La dificultad radica en esquivar aquellos objetos que nos restan energía, como los arbustos verdes (estáticos en cada pantalla) o las ranas que aparecen moviéndose en algunas de ellas.



Pero la verdadera gracia del juego está en la forma de manejar al personaje. Como resultado de su estado ebrio, el personaje no responde a nuestras órdenes como lo suelen hacer tradicionalmente los protagonistas de los videojuegos. En este caso la pulsación de las teclas direccionales (o bien el joystick) le imprime una aceleración en la dirección que marquemos, lo que dificulta considerablemente poder hacernos con el control y llevar al personaje exactamente por donde queremos.

Comenzamos la partida con 20 puntos de energía. Dichos puntos se pierden al contacto con los arbustos o las ranas. Por otro lado, cada pantalla tiene un tiempo límite para ser completada (esto es, haber rescatado a todos nuestros amigos). Si el tiempo acaba, perderemos un punto de vida. La partida concluirá cuando perdamos toda la energía. Eso sí, según vayamos recogiendo a nuestros amigos, podremos ir recuperando algo de la energía perdida.



Rescatando a nuestros amigos. La pelota amarilla somos nosotros

Gráficamente el juego no es ninguna maravilla. La cuestión radica en que el ritmo de juego es tan frenético que no tendremos tiempo de fijarnos en menudencias.

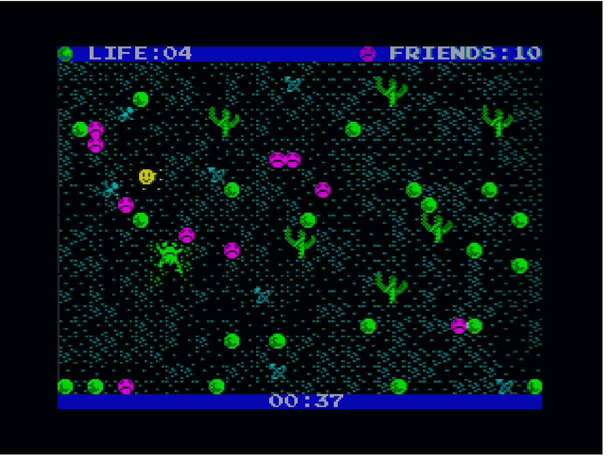
En cuanto al sonido, unas pegadizas melodías nos acompañarán en los menús, mientras que el juego en sí se ve aderezado por los típicos efectillos sonoros, que simplemente cumplen su cometido, sin ser nada destacables.

En cualquier caso se trata de un título que, si bien nunca será considerado un clásico ni técnicamente ni por su argumento ni su concepción, sí que trae un soplo de aire fresco a la escena y proporcionará un buen rato de diversión, intentando superarnos hasta que lo completemos.

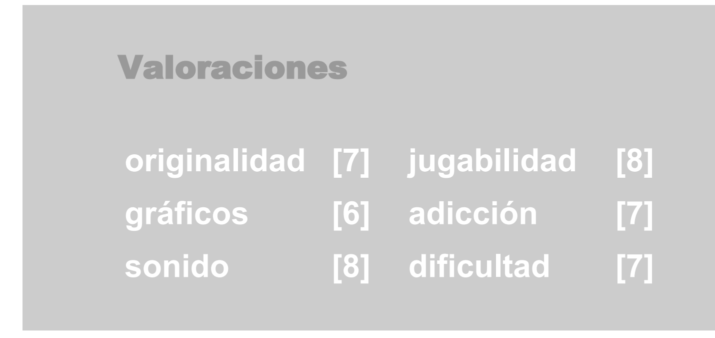
Es de agradecer a los autores, primero, que empleen su tiempo en crear juegos para el Spectrum y, segundo, que encima pongan a disposición de todo el mundo el código fuente. Este juego ha sido programado empleando el lenguaje C y el compilador Z88DK. Sin duda liberar el código fuente puede ayudar a que otros programadores se muestren interesados y se embarquen a su vez en la bonita aventura de crear un juego para la máquina de Sinclair.

descárgalo de

CEZGS (Computer EmuZone Games Studio)



Cuidadito con tocar todo lo que tenga color verde, arbustos o ranas



Valoraciones

originalidad	[7]	jugabilidad	[8]
gráficos	[6]	adicción	[7]
sonido	[8]	dificultad	[7]

FALVAREZ

al descubierto

Quizá EL BOMBAZO, con mayúsculas, de la pasada party MadriSX & Retro 2005 fue, sin lugar a dudas para los usuarios de Spectrum, la aparición de este juego inédito y más aun cuando fueron sus autores los que hicieron acto de presencia para presentárnoslo.

VEGA SOLARIS

Título	Vega solaris
Género	Maze
Año	1985-2005
Máquina	ZX Spectrum 48K
Jugadores	1 ó 2 Jugadores
Compañía	Quasar/Eclipse
Autor	Fernando Sáenz Pérez Carlos García Cordero

En este artículo, más que realizar un 'patas arriba' clásico, algo difícil de llevar a cabo ya que el programa es totalmente aleatorio y no hay una ruta única a seguir para su finalización, vamos a analizar el juego y después dar una serie de pistas que nos ayudarán a terminar con éxito nuestras andanzas.

Anunciado y defenestrado por Dinamic, Vega Solaris es una aventura épica, puesta en escena en un planeta extraño, en el que se baten en duelo un humano y un alienígena en busca del símbolo imposible de Vega Solaris. Con un argumento sencillo se desarrolla una compleja y difícil aventura. En nuestra tarea, conseguir recolectar las cuatro partes en las que está dividido el símbolo y llevarlas a la pantalla de incio, es el menor de los problemas:

El primer reto será el extenso mapeado del juego. Formado por doscientas once pantallas está dividido en tres zonas bien diferenciadas: comenzamos en el norte de un enorme templo que ocupa practicamente la mitad del mapa; al este del templo tenemos la selva y al sur una zona de grutas. La combinación de las pantallas forman un complicado laberinto que, si bien tiene muchos



puntos de referencia como para no perdernos (gracias a sus grandes gráficos) sí será adecuado el uso de un mapa para situarnos.

En segundo lugar en este laberinto se 'perderán' al azar las cuatro piezas del símbolo que tenemos que buscar y llevar, posteriormente, al punto de partida para completar nuestra misión.

Para complicar un poco más las cosas, cada habitación está repleta de enemigos que se lanzarán contra nosotros, a los cuales podemos disparar o, lo más sensato en muchas ocasiones, esquivar, ya que si perdemos el escaso tiempo de que disponemos para terminar la aventura, ésta finalizará.

Si bien el planteamiento de Vega Solaris no puede

ser tratado de original, ya que juegos como Sabre Wulf o Atic Atac utilizan un esquema similar, sí añade unas cuantas novedades que lo hacen muy adictivo y recomendable para jugar con un amigo... o enemigo. Nuestro contrincante, cuyo objetivo es el mismo que el nuestro, y sus ventajas y desventajas durante la partida serán las mismas, se moverá con soltura por el escenario del juego, si es que lo maneja el ordenador o puede ser manejado por otra persona, lo cual hace del juego un divertido enfrentamiento 'cara a cara', permitiéndonos incluso robar el contenido de sus bolsillos.



Pantalla de carga con el invaders en funcionamiento

Como ventajas tenemos una serie de hechizos dispersos por el escenario y que podemos recoger, los cuales una vez lanzados pueden cegar a nuestro oponente, lanzar bolas que eliminan a los enemigos, congelarlos o teletransportarnos a donde se encuentre el oponente. Estos hechizos tienen una duración limitada y si queremos coger un objeto, luchar a brazo partido o realizar cualquier otra tarea, tendremos que dejar de utilizarlos. También disponemos de armas: una espada, un arco y el eterno recurso del puñetazo. Debemos tener en cuenta que estas ventajas también las puede utilizar nuestro contrincante, y que nos puede dejar ciegos en cualquier momento de la partida, con el inconveniente que esto representa.

Llegados a este punto, el planteamiento del juego es claro. Para conseguir las cuatro piezas disponemos de espacio en nuestros bolsillo con capacidad para el mismo número de objetos. Con lo que cuando tengamos recogidas algunas se reducirá el número de armas y hechizos que podemos transportar y, por tanto, nos será más complicado defendernos. Como comentábamos con anterioridad, nuestro oponente persigue los mismos objetivos, para lo cual tendremos, en muchas ocasiones, que robarle las piezas que porta, o él intentará lo mismo. Para conseguir esto tenemos que aprovechar que nuestro enemigo esté insconciente, cosa que se consigue reduciendo su nivel de energía bien gracias al choque con los enemigos, o bien con nuestras armas, y acercándonos a él. Logicamente no sólo le podemos robar las piezas, también será posible 'levantarle' cualquier contenido del bolsillo.

El nivel de energía, indicado en el interface por un huevo que va cambiando de color, disminuye al contacto con los enemigos, o por el uso de armas del contrario. Cuando llega a cero no morimos, pero quedamos sin sentido durante unos segundos. Estos valiosos segundos pueden ser aprovechados por nuestro rival para robarnos, además de que el tiempo avanza inexorable, representado por una calavera con la parte superior en forma de bola del mundo. La única forma de que el juego termine es completando nuestra misión o agotando el tiempo. Por muchas veces que nuestra 'estamina' se agote, terminamos recuperándonos.

La presentación del juego en pantalla se divide en dos zonas, en cada una se representa la habitación que ocupa actualmente el personaje, y debajo de la zona de juego aparece el interface. En dicho interface, además de la vida restante, podemos ver el contenido de nuestros bolsillos y la función que tenemos seleccionada en ese momento, que puede ser una de estas cuatro: Coger o soltar objeto, usarlo, y cambiar el elemento seleccionado. Además se indica la puntuación y el tiempo restante, que se representa con la calavera y la bola del mundo; a más gráfico visto, menos tiempo para completar el juego.



Gráficos grandes y coloristas

Graficamente Vega Solaris es muy atractivo. Pese a que los juegos basados en 'tiles' pecan de ser muy repetitivos, en esta ocasión se han añadido unos gráficos de gran tamaño representando estatuas, columnas, extraños animales y algunas cosas más que hacen que destaque por encima de la media, y si añadimos el cambio entre las zonas que está dividido el mapa la nota roza la matrícula de honor. Los diferentes sprites, tanto de los enemigos como de los contrincantes, están bien resueltos, son definidos y claros.

El movimiento es rápido y veloz desplazándose los gráficos con fluidez por la pantalla de juego y respondiendo con gran celeridad a nuestros

deseos en forma de pulsaciones de teclado o manejo del joystick. Como controles podemos seleccionar diferentes modelos de joystick (Kempston, Sinclair o cursor) y redefinir las teclas. Todo ello desde un completo menú, desde el cual, además de seleccionar controles y definir las teclas, podemos elegir el personaje con el cual vamos a jugar o si el contrario lo controlará el ordenador o un amigo.

En el aspecto sonoro es donde el juego toca su nivel más bajo. No hay música ni en el menú ni durante el desarrollo de la partida y los efectos de sonido durante el juego cumplen sin más pretensiones.



Búsqueda en el laberinto

Como nota original, durante la carga del juego, con una pantalla diseñada por Azpiri, se puede jugar a un pequeño invaders para amenizar la espera. Un detalle curioso y que denota el cariño con el que se cuidó la realización del juego.

En resumen, técnicamente el juego alcanza niveles muy altos, y más si pensamos que se ha metido en apenas 40 kilobytes de memoria. Si bien este tipo de juego ya estaba en desuso en el año 1989, que es cuando se finalizó, creemos que en el año 1987, fecha en la que Dinamic lo publicitó en la revista MicroHobby hubiera supuesto un éxito de ventas y críticas, y más teniendo en cuenta el alto grado de inteligencia aplicado al manejo del enemigo, que en muchos casos parece manejado por un humano en lugar de por el ordenador, lanzando hechizos, recorriendo el mapa con lógica y tratando de completar el juego. El nivel de dificultad es bastante elevado al igual que la adicción que causa.

ALGUNOS TRUCOS E INDICACIONES

Antes decíamos que hacer una guía del juego para seguir a rajatabla es una tarea ardua. El juego reparte aleatoriamente tanto las cuatro partes del símbolo como los objetos por el mapeado, y no se puede seguir una constante a la hora de buscarlos. Por si esto fuera poco el enemigo puede recoger piezas y desplazarlas consigo o dejarlas en otras pantallas diferentes.

Un buen truco puede ser no utilizar los hechizos contra él hasta que no tenga piezas en su poder. Dejarlo ciego en ese momento o teletransportarnos hasta su localización puede reducir mucho el tiempo de búsqueda.

Básicamente nuestro gran enemigo es el tiempo. No sobra, es más, va demasiado justo para finalizar el juego. Aunque en alguna ocasión nos quedemos embobados observando algún gráfico (incluso hay uno bastante atrevido para la época de una mujer desnuda), lo mejor es utilizar un mapa y tenerlo bien estudiado para no dar vueltas en falso.

No perdamos el tiempo matando enemigos por las pantallas. La mejor opción es esquivarlos siempre que podamos, ya que se vuelven a regenerar con lo que no conseguiremos limpiar las pantallas por si tenemos que volver a pasar por ellas.

Os adjuntamos un mapa confeccionado por la redacción de Magazine ZX que, aunque ya lleva tiempo colgado en la WEB oficial del juego, siempre os será útil.

Lo demás lo dejamos para que investiguéis en el juego. Es un gran placer recorrer sus pantallas y descubrir como el enemigo nos hace la vida imposible con mucha coherencia.

Hay mucha más información, incluídos 'pokes', en la página oficial del juego:

http://vegasolaris.speccy.org/

Descarga el mapa de la aventura:

mapa_vega.zip

Valoraciones

originalidad	[8]	jugabilidad	[10]
gráficos	[9]	adicción	[10]
sonido	[6]	dificultad	[9]

MIGUEL

hardware

RESEÑA DEL ADAPTADOR IDE A COMPACT FLASH DE JOSÉ LEANDRO

Uno de los principales problemas a la hora de disfrutar del Spectrum es la tediosa carga desde la cinta de cassette. Si somos los felices poseedores de un +3, los tiempos de carga se alivian bastante. Sin embargo, se trata de un modelo que no estuvo muy extendido en su momento (ya que era competencia directa del CPC6128), y en la actualidad se cotizan bastante caros. En nuestro país el modelo más extendido quizás sea el +2A/B.

Hace tiempo que Garry Lancaster dio forma al proyecto +3E, que consiste en una ROM sustituta de las originales de los modelos +2A, +2B y +3 en la que se incluyen nuevos comandos y la capacidad de manejar dispositivos IDE de una forma sencilla. Así, podemos conectar un disco duro a nuestro Spectrum mediante una interfaz muy sencilla de construir, y acabaremos con nuestros problemas de velocidad de carga (y, por descontado, de espacio de almacenamiento).



El adaptador IDE a Compact Flash de José Leandro

El problema es que incorporar un disco duro externo al Spectrum no es muy manejable. Para empezar, necesitaremos una fuente de alimentación extra a la que conectar el disco duro. Si usamos una fuente de alimentación AT o ATX, que suele ser la solución más corriente, tendremos que soportar el ruido infame del ventilador (uno de los encantos del Spectrum es su silencio cuando está encendido). A ello se añade el cable IDE y el propio disco duro en si, que también ocupa su espacio.

José Leandro Novellón, todo un mago de la electrónica aplicada al Spectrum, tuvo esas mismas inquietudes, así que se puso a buscar una solución alternativa.

Lo primero que hizo fue adquirir un lector IDE de tarjetas Compact Flash. Como hemos comentado anteriormente, gracias a las ROMS del +3E, el Spectrum puede manejar dispositivos IDE. La cuestión viene a la hora de usar la tarjeta, ¿con qué parametros hay que formatearla? Cuando se trata de un disco duro podemos ver la información de número de cabezas, cilindros y sectores normalmente impresa en el propio disco, pero en el caso de las tarjetas esto no es así.

Conectando el adaptador IDE al PC se puede leer de la BIOS el valor de dichos parámetros. Pese a todo, parece no funcionar bien, así que habrá que ajustarlos un poco más hasta dar con los correctos.

Una vez comprobado que el invento funcionaba, el siguiente reto fue diseñar una placa específica que aunara el interfaz IDE y el adaptador para Compact Flash. Un reto para cualquier mortal, pero para José Leandro esta es la parte en la que se encuentra en su salsa.



El adaptador conectado al +2E

Conectando el adaptador IDE al PC se puede leer de la BIOS el valor de dichos parámetros. Pese a todo, parece no funcionar bien, así que habrá que ajustarlos un poco más hasta dar con los correctos.

Una vez comprobado que el invento funcionaba, el siguiente reto fue diseñar una placa específica que aunara el interfaz IDE y el adaptador para Compact Flash. Un reto para cualquier mortal, pero para José Leandro esta es la parte en la que se encuentra en su salsa.

FALVAREZ

LINKS

Artículo escrito por José Leandro para El Trastero del Spectrum

Proyecto +3E

ZXMINES: UN JUEGO EN BASIC COMENTADO

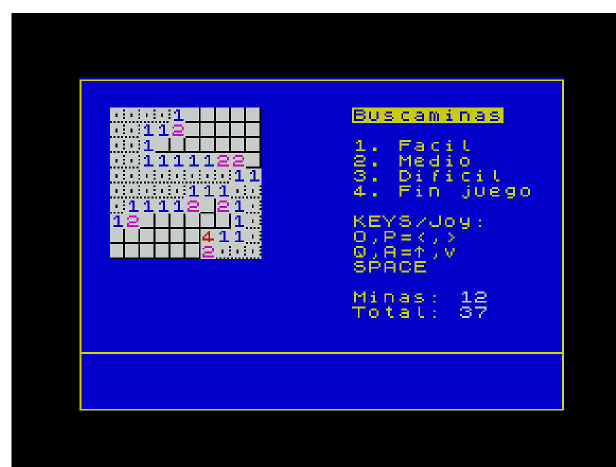
ZXMines es un juego en BASIC que se presentó al concurso de programación de Juegos en Basic 2003 de ByteManiacos. ZXMines implementa en BASIC un sencillo juego de Buscaminas en el cual debemos destapar todo el tablero sin levantar las casillas en que se alojan las minas (destapar una mina finaliza el juego). Para poder destapar totalmente el tablero sin levantar casillas con minas disponemos de una información providencial: cada casilla que no contiene una mina nos indica numéricamente cuántas minas hay en las 8 casillas de alrededor de la casilla actual.

Así, si destapamos una casilla y contiene el número 1, sabemos que alguna de las 8 casillas de alrededor de la misma contiene una mina. Utilizando la información numérica que nos proporcionan las diferentes casillas que vamos destapando podemos ser capaces de averiguar qué casillas contienen minas y evitarlas. El juego acaba cuando destapamos una mina (y perdemos) o bien cuando destapamos todas las casillas del tablero quedando sólo por destapar las casillas con minas (ganando el juego). Por último, una cosa a destacar es que si destapamos una casilla sin minas alrededor, se abre todo un área de juego a la vista, para acelerar el ritmo de juego.

El objetivo del presente artículo es mostrar y explicar el código BASIC utilizado para programar ZXMINES, mostrando así algunos trucos que en BASIC proporcionan una mayor velocidad de ejecución.

PSEUDOCÓDIGO DEL JUEGO

Lo primero que hacemos es definir el juego mediante lenguaje humano, para posteriormente adaptar ese lenguaje humano a código en BASIC. Es muy importante hacer esto antes de escribir una sóla línea en BASIC. El programa se adapta al diseño y al pseudocódigo, y no al revés.



ZXMines: el clásico juego Buscaminas, en BASIC

Veamos el pseudocódigo para nuestro juego buscaminas:

```
Inicio ZXMINES
 - Inicio del programa (declaracion de variables, etc.).
 - Dibujado de la pantalla
 - Bucle principal del juego:
   - Comprobacion de fin de partida:
     - Si quedan tantas casillas como minas:
       - Llamar a Victoria_Fin_Juego
```

```
  - Leer teclado
     - Si tecla es 1, 2, ó 3:
        - Cambiar la dificultad
        - Nueva partida, saltando a "Dibujado de la pantalla"
     - Si tecla es 4 Entonces Fin del juego.
     - Si tecla es ARRIBA:
         - Mover cursor arriba (ycursor = ycursor-1)
     - Si tecla es ABAJO:
         - Mover cursor abajo (ycursor = ycursor+1)
     - Si tecla es IZQUIERDA:
         - Mover cursor izquierda (xcursor = xcursor-1)
     - Si tecla es DERECHA:
         - Mover cursor derecha (xcursor = xcursor-1)
     - Si tecla es DISPARO:
         - Destapar la casilla bajo (xcursor,ycursor)
           llamando a DestaparMina


FUNCIÓN DestaparMina:
 - Si debajo de (xcursor,ycursor) hay una casilla por destapar:
   - Si debajo de (xcursor,ycursor) hay una mina:
     Muerte_Fin_Juego
   - Si debajo de (xcursor,ycursor) no hay mina:
     Blanquear_Cuadros_Alrededor
   - Actualizar puntos y numero de casillas y minas restantes.


FUNCIÓN Muerte_Fin_Juego:
 - Imprimir por pantalla mensajes, esperar tecla.
 - Restart del juego.


FUNCIÓN Victoria_Fin_Juego:
 - Imprimir por pantalla mensajes, esperar tecla.
 - Restart del juego.


FUNCIÓN Blanquear_Cuadros_Alrededor:
 - Rutina que recibe una posición CX,CY y destapa
   todas las casillas no minas a partir de CX,CY.


FUNCIÓN Generacion_de_tablero:
 - Definir una array bidimensional de 12x12 (dificultad maxima)
   para almacenar las minas (V).
 - Definir una array bidimensional de 12x12 (dificultad maxima)
   para almacenar si una casilla está destapada o no, 0/1 (T).
 - Poner a cero todos los elementos de los arrays.
 - Segun la dificultad actual, llenar el tablero con minas de
   forma aleatoria (tablero de 8x8, 10x10 o 12x12). Las minas
   se colocan en V() mediante un número "9".
 - Actualizar todas las casillas del tablero recontando el
   número de minas alrededor, y colocando ese número en la
   casilla correspondiente.

FUNCIÓN Preparar_Pantalla:
 - Limpiar pantalla.
 - Imprimir textos, título, teclas.
 - Definir ciertas variables (minas, dificultad, tamaño tablero)
 - Cargar los GDUs/UDGs desde los DATAs.

Fin ZXMINES
```

El pseudocódigo permite implementar el juego en cualquier lenguaje de una manera rápida: por ejemplo, el mismo pseudocódigo/diseño que se usó para ZXMINES 1 en BASIC se utilizó para programar ZXMINES 2 en C (mucho más rápido) en apenas un par de horas de codificación. Para que el pseudocódigo sea legible es recomendable mantenerlo conciso y no extenderse demasiado en detalles, ya que debe ser una visión general del programa.

EL LISTADO DEL JUEGO

En los siguientes apartados veremos los diferentes bloques funcionales del juego comentados. El lector podrá identificar fácilmente a qué función del pseudocódigo se corresponden. Como puede verse, la utilización de pseudocódigo permite una programación más limpia, ya que la implementación sólo tiene que ceñirse a lo que hemos diseñado, reduciendo los errores posteriores (se trata tan sólo de implementar la visión general que nos da le pseudocódigo, y esto se puede hacer de forma modular).

El código lo programaremos en un editor de texto convencional en nuestro ordenador personal y lo grabaremos como un simple fichero de texto con extensión .bas. Después, con bas2tap (un utilísimo programa) generaremos un TAP que será equivalente a haber tecleado el programa en el intérprete BASIC del Spectrum y haberlo grabado con SAVE. El tap resultante de la "compilación" se podrá ejecutar tanto en emuladores como en Spectrum reales.

Para empezar, veamos la relación entre el pseudocódigo y las diferentes líneas BASIC del programa (podéis utilizar el Listado 1 para identificar los diferentes bloques funcionales):

```
Inicio ZXMINES
 - Inicio del programa (líneas 48-90)
 - Dibujado de la pantalla (líneas 105-140 y 2300-9022)
 - Bucle principal del juego: (lineas 200-990)

FUNCIÓN DestaparMina: (líneas 1000-1099)
FUNCIÓN Muerte_Fin_Juego: (líneas 1300-1330)
FUNCIÓN Victoria_Fin_Juego: (líneas 1600-1630)
FUNCIÓN Blanquear_Cuadros_Alrededor: (líneas 3-14)
FUNCIÓN Generacion_de_tablero: (líneas 2000-2200)
FUNCIÓN Preparar_Pantalla: (líneas 105-140 y 2300-9022)
Fin ZXMINES
```

Como puede verse, cada función o bloque lógico del pseudocódigo se corresponde con un bloque de líneas BASIC, que será lo que veremos más detallado a continuación.

INICIO DEL PROGRAMA Y PREPARACIÓN DE PANTALLA

Aparte de los comentarios del programa (líneas 9100 a 9147), el programa (lógicamente hablando) empieza con el siguiente código:

```
48 INPUT "Spanish/English? (s/e)", L$ ;
49 IF L$<>"s" AND L$<>"S" AND L$<>"e" AND L$<>"E" THEN GO TO 48

50 REM *** Declaracion de las variables del programa ***
51 LET COL=6 : LET PAP=1
53 INK COL: PAPER PAP
54 IF L$="E" THEN LET L$="e"
55 IF L$="S" THEN LET L$="s"
60 DIM T(12,12)          : REM Campo de minas
65 DIM V(12,12)          : REM Visto o no visto (0/1)
70 LET BUCLE=1
80 LET IX=1 : LET IY=1    : REM IX, IY del tablero
85 LET DIFICULTAD=0
90 GO SUB 8000           : REM DEFINIMOS LOS GRAFICOS
```

Lo primero que hacemos es pues preguntar al usuario el idioma para el juego, y después declarar las variables que utilizaremos en el programa. La variable L$ almacenará el idioma del juego, pudiendo contener la cadena "e" (english) o la cadena "s" (spanish). La usaremos posteriormente a la hora de hacer PRINTs para imprimir mensajes en un idioma u otro.

Dos variables importántisimas son los arrays bidimensionales T(12,12) y V(12,12), que se utilizarán para representar el tablero de juego del buscaminas. Se definen de 12x12 porque éste es el tamaño máximo del juego para el nivel de dificultad máxima; en caso de utilizar menores niveles de dificultad (8x8 y 10x10) se utilizará el mismo array de 12x12, pero usando sólo una porción del mismo.

El array V(12,12) nos indica el estado de las casillas indicando si están tapadas (0), o si están destapadas (1), es decir, si vemos el contenido de la casilla o no. Inicialmente este array tendrá todos sus elementos a cero porque al comenzar el juego todas las casillas están tapadas.

Así pues, un tablero tapado tendría la siguiente forma:

```
V =
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
```

Un tablero con la casilla central destapada tendría el siguiente aspecto:

```
V =
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,1,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
```

El array T(12,12) indica lo que contiene cada casilla propiamente dicha. Si contiene un cero, la casilla está vacía. Si contiene un nueve, la casilla tiene una mina. Finalmente, cualquier valor entre 1 y 8 significa el número de minas que hay alrededor de la casilla actual.
Así, inicialmente tendremos ambos arrays (V y T) llenos de ceros porque no hay minas dentro, y porque todas las casillas están tapadas. Si queremos introducir una mina en el tablero en la posición (3,5), podemos hacerlo con:

```
T(3,5) = 9;
```

Al igual que en el caso anterior, un array sin minas estará lleno de ceros, pero un array con 4 minas y ya correctamente "rellenado" (como veremos posteriormente), podría tener un aspecto similar al siguiente:

```
T =
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,1,1,1,0,0,0,0,0
0,0,0,0,1,9,1,0,0,0,0,0
0,0,0,0,1,1,1,0,0,0,0,0
0,0,0,0,0,0,0,0,1,1,1,0
1,1,1,0,0,0,0,0,2,9,2,0
1,9,1,0,0,0,0,0,2,9,2,0
1,1,1,0,0,0,0,0,1,1,1,0
0,0,0,0,0,0,0,0,0,0,0,0
```

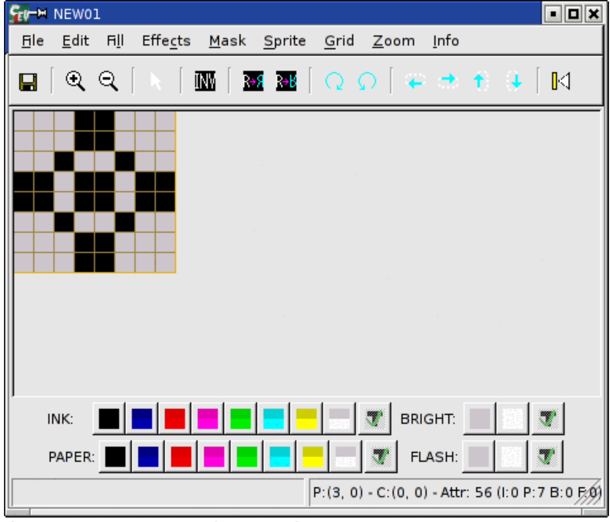
Como puede verse, las minas se sitúan con números 9, y las casillas de alrededor de las minas contienen valores numéricos que indican la cantidad de minas circundantes.

Finalmente se produce un salto a la rutina de la dirección 8000, que prepara los UDG o gráficos definidos por el usuario:

```
8000 REM *** Preparacion de GDUs
***
8005 RESTORE 9012
8010 FOR I=0 TO 7: READ FILA :
POKE USR "I"+I, FILA : NEXT I
8015 FOR I=0 TO 7: READ FILA :
POKE USR "M"+I, FILA : NEXT I
8020 FOR I=0 TO 7: READ FILA :
POKE USR "E"+I, FILA : NEXT I
```

Como puede verse, se define "M" como el UDG que contiene el dibujo de las minas, "E" como un UDG que contiene el dibujo de una casilla sin destapar o esquina, e "I" como el UDG que contiene el dibujo de una casilla destapada vacía (un pequeño puntito central).

Los 3 dibujos han sido creados con SevenuP de la misma forma. Por ejemplo, para crear la mina, abrimos SevenuP y creamos un nuevo dibujo de 8x8. Dibujamos la mina (en la siguiente figura la veremos a medio dibujar) y vamos a File , y luego a Output Options y seleccionamos "No attributes" y como Byte Sort Priority dejamos "X char, Char line, Y char, Mask".



Creando nuestros gráficos en SevenuP

A continuación exportamos los datos, desde el menu File, opción Export Data. Escribimos un nombre de fichero (por ejemplo, mina.c), y en el desplegable seleccionamos "C source" en lugar de "raw Binary". Al darle a OK habremos creado un fichero mina.c con el gráfico exportado a formato C. El aspecto del fichero será similar (comentarios aparte) al siguiente:

```
unsigned char Sprite[8] = {16,
186, 116, 254, 124, 186, 16, 0 };
```

Estos son los valores que podremos en nuestros DATA en BASIC:

```
9017 DATA 16, 186, 116, 254,
 124, 186, 16, 0
```

Tras el inicio del programa (líneas 48 a 90) viene la preparación de la pantalla (líneas 2300 a 2450) donde simplemente se imprimen en pantalla los mensajes que veremos en el menú principal.



Aspecto de la pantalla de presentación

GENERACIÓN DEL TABLERO

La rutina de las líneas 2000 a 2200 es llamada desde la rutina de preparación de la pantalla para rellenar el tablero T con minas aleatorias (números 9), y al mismo tiempo calcular los valores numéricos de cada casilla indicando el número de minas alrededor.

La rutina viene a hacer algo como lo siguiente:

- Rellenar el tablero con minas aleatorias.
- Recalcular el valor numerico de cada casilla sin mina para todo el tablero.

En pseudocódigo:

```
Desde I = 0 a NUMERO_DE_MINAS
   Poner minas aleatoriamente con T(RANDOMX,RANDOMY) = 9;
 Fin Desde

 Desde Y = 0 a ALTO_TABLERO
   Desde X = 0 a ANCHO_TABLERO
     Contar el numero de minas alrededor de T(x,y)
     T(x,y) = ese numero de minas
   Fin Desde
 Fin Desde
```

El código resultante es:

```
2000 REM *** GENERACION DE TABLERO ***
2001 IF DIFICULTAD=0 THEN RETURN
2002 IF L$="e" THEN PRINT AT 19,1 ; "Generating board..."; AT 20,1 ;
     "> Placing mines: 0%";
2003 IF L$="s" THEN PRINT AT 19,1 ; "Generando tablero.."; AT 20,1 ;
     "> Insert. minas: 0%";
2030 LET INCR=100/MINAS : LET PORCEN=0 : LET CON=1
2040 REM Primero ponemos a cero todo el tablero
2042 DIM T(12,12) : DIM V(12,12)
2045 LET RX = INT (RND*(ANCHO))+1
2046 LET RY = INT (RND*(ALTO))+1
2047 REM Ahora instalamos las minas. Si encontramos un hueco, pasamos a
     por otra mina
2050 FOR N=1 TO MINAS
2060   IF V(RX,RY)=1 THEN GO TO 2070
2065     LET V(RX,RY)=1 : POKE (59000+N),RX : POKE (59300+N),RY
2067     LET CON=CON+1 : IF CON=5+DIFICULTAD THEN PRINT AT 20,18 ;
     INT PORCEN ; "%"; : LET CON=1
2068     LET PORCEN=PORCEN+INCR : NEXT N
2069   REM Si no, generamos otro numeros y dec el indice
2070   LET RX = INT (RND*(ANCHO))+1
2080   LET RY = INT (RND*(ALTO))+1
2090   LET N = N-1
2100 NEXT N
2101 IF L$="e" THEN PRINT AT 20,1 ; "> Preparing board: 0%";
2102 IF L$="s" THEN PRINT AT 20,1 ; "> Gener.  tablero: 0%";
```

En esa primera parte colocamos las minas dentro del tablero, y a continuación calculamos los valores numéricos de cada celdilla del tablero:

```
2109 LET PORCEN=0 : LET CON=0
2110 FOR N=1 TO MINAS
2111 LET CON=CON+1 : IF CON=5+DIFICULTAD THEN PRINT AT 20,20 ;
     INT PORCEN ; "%"; : LET CON=1
2112   LET PORCEN=PORCEN+INCR
2120   LET X=PEEK (59000+N) : LET Y=PEEK (59300+N)
2131   IF X=1 THEN GO TO 2138
2132     LET T(X-1,Y) = T(X-1,Y) + 1
2133     IF Y>1          THEN LET T(X-1,Y-1)=T(X-1,Y-1)+1
2134     IF Y1          THEN LET T(X+1,Y-1)=T(X+1,Y-1)+1
2145   IF Y1    THEN LET T(X,Y-1)=T(X,Y-1)+1
2160 NEXT N
2161 PRINT AT 20,20 ; "100%";
2165 REM Ahora plantamos las minas
2170 FOR N=1 TO MINAS : LET T( PEEK (59000+N), PEEK (59300+N)) = 9 :
     NEXT N
2175 DIM V(12,12)
2180 PRINT AT 19,1 ; INK COL; PAPER PAP; "                    ";
2181 PRINT AT 20,1 ; INK COL; PAPER PAP; "                        ";
2200 RETURN
```

Nótese un pequeño truco que se ha utilizado para acelerar la generación y cálculos: en lugar de trabajar sobre arrays temporales, trabajamos sobre zonas de memoria (59000 y 59300) tratándolas como vectores de trabajo temporales. Escribimos en nuestros "arrays" en memoria con POKE, y leemos con PEEK. Posteriormente en la rutina de "Blanqueado de casillas" veremos su utilidad y porqué se ha realizado esto en lugar de arrays temporales. En las posiciones de memoria desde 59000 a 59000+numero_de_minas (ese numero_de_minas varía con la dificultad de juego elegida) se almacenan las posiciones X de las minas aleatorias generadas. Lo mismo ocurre en 59300, donde se almacenan las posiciones Y de las minas aleatorias generadas.

Así, la primera parte de la función coloca en 59000 y 59300 valores X e Y para las minas, y finalmente incorporamos las minas al tablero mediante:

```
2170 FOR N=1 TO MINAS : LET
T(PEEK (59000+N), PEEK (59300+N))
= 9 : NEXT N
```

El resultado de la ejecución de la rutina de arriba es un tablero tapado (V(x,y) = 0 para todo x e y), y que tiene una serie de minas (valores numéricos 9) en T(xminas,yminas), estando el resto de casillas de T(x,y) calculadas con valores numéricos que representan las minas alrededor de la casilla en cuestión, como en el ejemplo que hemos visto antes:

```
T =
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,1,1,1,0,0,0,0,0
0,0,0,0,1,9,1,0,0,0,0,0
0,0,0,0,1,1,1,0,0,0,0,0
0,0,0,0,0,0,0,0,1,1,1,0
1,1,1,0,0,0,0,0,2,9,2,0
1,9,1,0,0,0,0,0,2,9,2,0
1,1,1,0,0,0,0,0,1,1,1,0
0,0,0,0,0,0,0,0,0,0,0,0
```

Una vez tenemos definido el tablero de juego podemos continuar con el bucle principal del programa.

BUCLE PRINCIPAL DEL JUEGO

El bucle principal del juego se desarrolla entre las líneas 200 y 990).

Lo primero que hacemos es dibujar el tablero de juego con 2 bucles FOR para el ancho y para el alto:

```
200 REM *** BUCLE PRINCIPAL ***
300 LET P$= CHR$(16) + CHR$(0) +
    CHR$(17) + CHR$(7) :
    FOR X=1 TO ANCHO : LET
    P$=P$+"{E}" : NEXT X
310 FOR Y=1 TO ALTO
315 PRINT AT IY+Y,IX+1 ; P$;
370 NEXT Y
```

Se ha utilizado una cadena "P$" en la que introducimos toda una fila horizontal de casillas sin destapar (E). Después con un bucle pintamos tantas líneas horizontales de casillas como altura tiene nuestro tablero. Esto es más rápido que un doble bucle, donde tendríamos que recalcular las casillas horizontales cada vez. El símbolo "{E}" representa al UDG "E" (para bas2tap).

A continuación viene el bucle principal en sí mismo. Lo que se hace en este bucle principal es utilizar unas variables CX, CY que indican la posición dentro del array de la casilla "actual" o cursor que estamos manejando (de forma que cuando pulsamos espacio destapamos la casilla actual). Las variables CX y CY (posición de la casilla/cursor) se actualizan cuando pulsamos izquierda, derecha, arriba o abajo. En cada paso del bucle representamos la casilla "actual" o cursor mediante un FLASH de la posición en pantalla de la casilla (IX+CX,IY+CY). Recordad: la posición (CX, CY) es un valor entre 1 y 12 dentro de los arrays V() y T(), mientras que (CX+IX, CY+IY) representa un valor entre 1-32 y 1-24 para dibujar en pantalla el cuadro parpadeante (IX e IY son el inicio en pantalla del tablero).

```
400 REM Bucle principal de partida
405 IF DIFICULTAD=0 THEN GO TO 415
407 IF MINAS=QUEDAN THEN GO TO 1600
410 OVER 1: PRINT AT IY+CY, IX+CX ; FLASH 1; INK 0 ; PAPER 7;" "; :
    OVER 0 : GO SUB 991
415 LET K$=INKEY$ : IF K$="" THEN GO TO 415
421 IF K$="1" THEN LET DIFICULTAD=1 : BEEP .0015,35 : GO SUB 2400 :
    GO SUB 1460 : GO SUB 996 : GO
TO 111
422 IF K$="2" THEN LET DIFICULTAD=2 : BEEP .0015,35 : GO SUB 2400 :
    GO SUB 1460 : GO SUB 996 : GO
TO 111
423 IF K$="3" THEN LET DIFICULTAD=3 : BEEP .0015,35 : GO SUB 2400 :
    GO SUB 1460 : GO SUB 996 : GO
TO 111
424 IF DIFICULTAD=0 THEN GO TO 990
425 IF K$="4" AND DIFICULTAD<>0 THEN LET DIFICULTAD=0 : BEEP .0015,35 :
    GO TO 1300
```

```
430 IF K$="o" OR K$="O" OR K$="6" THEN IF DIFICULTAD<>0 THEN
    IF CX>1 THEN GO SUB 1500 : LET CX=CX-1 : GO TO 500
440 IF K$="p" OR K$="P" OR K$="7" THEN IF DIFICULTAD<>0 THEN
    IF CX<>0 THEN
    IF CY>1 THEN GO SUB 1500 : LET CY=CY-1 : GO TO 500
460 IF K$="a" OR K$="A" OR K$="8" THEN IF DIFICULTAD<>0 THEN
    IF CY<>0 THEN GO SUB 1000
990 GO TO 400
```

La actualización de la posición actual del cursor se realiza en la línea 410:

```
410 OVER 1: PRINT AT IY+CY, IX+CX ; FLASH 1; INK 0 ; PAPER 7;" "; :
    OVER 0 : GO SUB 991
```

Al final de esa línea se salta a la rutina 991, que simplemente se encarga con varios PRINT de actualizar los valores numéricos de números de minas y casillas restantes en el juego:

```
991 REM *** Actualizar contadores minas ***
992 IF L$="e" THEN PRINT AT 14, 18; "Mines: "; INK 7 ; MINAS ;  " "
993 IF L$="s" THEN PRINT AT 14, 18; "Minas: "; INK 7 ; MINAS ;  " "
994 IF L$="e" THEN PRINT AT 15, 18; "Cells: "; INK 7 ; QUEDAN ; " " :
    RETURN
995 PRINT AT 15, 18; "Total: "; INK 7 ; QUEDAN ; " " : RETURN
996 PRINT AT 14, 18; "           "
997 PRINT AT 15, 18; "            "
998 RETURN
```

Como puede verse, en el bucle principal también se controla la pulsación de las teclas 1 a 4 para cambiar la dificultad y acabar el juego. Nótese también la siguiente línea:

```
407 IF MINAS=QUEDAN THEN GO TO 1600
```

La línea 407 es el control de fin de juego para saber si el jugador ha ganado: si quedan tantas casillas por destapar como minas hemos puesto en el tablero, quiere decir que hemos destapado todas las casillas del tablero excepto las minas, con lo cual el juego se ha acabado y hemos ganado. En ese caso se salta a la línea 1600 que es la rutina de Victoria_Fin_Juego. Las rutinas de Victoria_Fin_Juego y Muerte_Fin_Juego son bastante sencillas (líneas 1300-1330 y 1600-1630): simplemente muestran un mensaje en pantalla, muestran el tablero desplegado y completo (si pulsas una mina y pierdes, tienes derecho a ver el contenido del tablero y la posición de las minas), esperan la pulsación de una tecla, reinicializan las variables del juego y saltan a la línea 400 (Inicio del bucle principal) para comenzar una nueva partida. La muestra y borrado del tablero se realiza mediante las siguientes rutinas:

```
1400 REM *** Mostrar tablero ***
1420 FOR Y=1 TO ALTO
1425   LET P$= CHR$(16) + CHR$(0) + CHR$(17) + CHR$(7) :
     FOR X=1 TO ANCHO : LET P$=P$+C$(T(X,Y)+1) : NEXT X  :
     PRINT AT IY+Y,IX+1 ; P$;
1450 NEXT Y
1451 RETURN

1460 REM *** Borrar tablero ***
1461 FOR N=1 TO 12
1462 PRINT AT IY+N,IX ; INK COL; PAPER PAP ; "                ";
1463 NEXT N
1470 RETURN
```

El borrado se basa simplemente en pintar espacios sobre el tablero, mientras que la visualización del tablero muestra todos los valores de T(X,Y) (independientemente del valor de V(X,Y), ya que pretendemos mostrar el tablero destapado completo). Puede verse el uso de C$(), que es un array donde hemos almacenado los caracteres para los 10 valores posibles del tablero (0 a 9, desde una casilla sin mina hasta una mina), de tal forma que 0 se corresponde con una casilla vacía, los números del 1 al 8 con los caracteres del 1 al 9, y el 9 con una mina.

```
101 DIM C$(10) : FOR N=1 TO 10: READ C$(N) : NEXT N
102 DIM H(10) : FOR N=1 TO 10: READ H(N) : NEXT N
103 DATA "{I}","1","2","3","4","5","6","7","8","{M}"
104 DATA 0, 1, 3, 2, 2, 2, 2, 2, 2, 0
```

La notación {M} significa "el UDG de M mayúscula", y se utiliza en BAS2TAP para especificar los UDGs (ya que los teclados de los PCs no tienen las teclas de nuestro Spectrum para dibujar esos símbolos).

Por último, cuando movemos el "cursor" de una posición a otra tenemos que recuperar en pantalla lo que había bajo la posición que abandonamos. Esta tarea la realiza la siguiente función:

```
1500 REM *** Restaurar grafico bajo el cursor ***
1510 IF V(CX,CY)=1 THEN PRINT AT CY+IY,CX+IX ; FLASH 0; PAPER 7;
     INK H(T(CX,CY)+1) ;C$(T(CX,CY)+1); : RETURN
1520 PRINT AT CY+IY,CX+IX; FLASH 0; INK 0 ; PAPER 7; "{E}";
1530 RETURN
```

De nuevo podemos ver cómo se hace uso de CX+IX y CY+IY para saber dónde "escribir" la casilla que hay que recuperar, y de C$() para saber qué carácter imprimir, y H() para saber con qué atributo (tinta) imprimirlo.

Gracias a H(), por ejemplo, podemos hacer:

```
102 DIM H(10) : FOR N=1 TO 10: READ H(N) : NEXT N
104 DATA 0, 1, 3, 2, 2, 2, 2, 2, 2, 0
(...)
PRINT INK H( valor ) ; "X" ;
```

en lugar de:

```
IF valor="0" : PRINT INK 0; "X"
IF valor="1" : PRINT INK 1; "X"
IF valor="2" : PRINT INK 3; "X"
(...)
IF valor="9" : PRINT INK 0; "X"
```

Lo cual es mucho más rápido y más óptimo y legible.

DESTAPADO DE MINAS

Cuando en el bucle principal pulsamos espacio, se salta a una rutina que destapa la casilla actual:

```
470 IF K$=" " OR K$="0" THEN IF DIFICULTAD<>0 THEN GO SUB 1000
```

Dicha rutina DestaparMina se desarrolla desde las líneas 1000 a 1099 del programa: Si la casilla ya está destapada, no hay nada que destapar. En caso contrario, la marcamos como ya abierta, y reducimos el número de casillas, y en una tercera línea imprimimos en pantalla el carácter al que corresponde la casilla que acabamos destapar:

```
1000 REM *** Destapar mina ***
1010 IF V(CX,CY)=1 THEN RETURN
1020 LET V(CX,CY)=1 : LET QUEDAN=QUEDAN-1 : BEEP .0005,35 :
1030 PRINT AT CY+IY,CX+IX ; PAPER 7; INK H(T(CX,CY)+1) ;
     C$(T(CX,CY)+1) ;
```

Si es una mina lo que hemos destapado, hemos perdido. Si, por contra, es una casilla en blanco, tenemos que destapar todas las casillas en blanco alrededor de la casilla actual, algo que se hace en la porción de código a partir de la línea 1060. Para el resto de los casos (números 1 al 8) basta con haber destapado y mostrado la casilla, y podemos volver (saltando a 1098):

```
1040 IF T(CX,CY)=9 THEN GO TO 1300
1050 IF T(CX,CY)=0 THEN GO TO 1060
1055 GO TO 1098
```

A partir de aquí empieza el destapado de recuadros en blanco: lo primero es visualizar un mensaje de información de que se va a saltar a un cálculo que en BASIC es algo lento y costoso, para después llamar a la rutina Blanquear_Cuadros_Alrededor (llamada en línea 1075). Tras volver de la rutina se actualizan los contadores de minas, casillas y puntos y se vuelve al bucle principal.

```
1060 REM Destapar todos los cuadros de alrededor que sean blancos
1066 LET PP=1                       : REM Puntero a la pila
1069 LET PD=1                       : REM Puntero a casillas destapadas
1070 IF L$="e" THEN PRINT AT 19,9 ; "...Working...";
1071 IF L$="s" THEN PRINT AT 19,9 ; "Calculando...";
1075 LET OX=CX: LET OY=CY : GO SUB 3 : LET CX=OX : LET CY=OY
1080 FOR N=1 TO PD-1
1090 LET X=PEEK (59000+N) : LET Y=PEEK (59300+N) :
     PRINT AT Y+IY,X+IX ; PAPER 7; INK H(T(X,Y)+1) ; C$(T(X,Y)+1) ;
1095 NEXT N
1096 LET QUEDAN=QUEDAN-PD+1 : LET PUNTOS=PUNTOS+PD-2
1097 PRINT AT 19,9 ; "               ";
1098 LET PUNTOS=PUNTOS+1
1099 RETURN
```

La rutina Blanquear_Cuadros_Alrededor se encarga de destapar todos los recuadros en blanco si el recuadro que acabamos de destapar está en blanco. Se usa para evitar que en zonas grandes de recuadros sin minas ni números tengamos que destapar uno a uno todos los recuadros que no contienen nada. Si no se llamara a esta función, al pulsar en un recuadro de un área vacía, simplemente se destaparía ese recuadro y ningún otro más. Lo que hace esta función es destapar todo el área vacía, para agilizar el juego.
Esta rutina se ha implementado aparte y de una forma optimizada por razones de velocidad. Para empezar, se ha situado en las líneas del 2 al 14 porque las primeras líneas del programa son las que BASIC ejecuta con mayor velocidad (pequeños trucos de BASIC). Además, se ha intentado reducir el número de líneas agregando muchos comandos a cada línea mediante ":", ya que eso también acelera la ejecución (digamos que el pasar de una línea a la siguiente requiere un tiempo en BASIC, y acumulando comandos en la misma línea lo reducimos). Como es la función más crítica (y lenta) del programa, se ha implementado aprovechando estos truquillos de BASIC, para acelerarla. Aún así hay que decir que tal y como está escrita la función es lenta (y esto es una invitación al lector a escribir una rutina más rápida, aprovechando la modularidad del programa).

Por último, esta función tiene una optimización extra que ha añadido algo de velocidad a la ejecución de la misma: se han cambiado los accesos al array (V(x,y) = valor) por accesos a memoria (con PEEK y POKE), al igual que se ha hecho en la rutina de generación del tablero y posicionamiento de las minas.

Lo que se hace, al igual que en el caso de la generación del tablero, es usar buffers de memoria como vectores de trabajo, empleando PEEK y POKE. Tras las pruebas realizadas, resulta que A(10)=1 es más lento que POKE 59000+10,1 . Así pues, en las direcciones 59000, 59300, 59500 y 59700 establecemos 4 buffers temporales donde trabajar.

El algoritmo de la rutina de las líneas 3 a 14 lo que hace es lo siguiente:

- Si la casilla actual es cero, destaparla (V(X,Y) = 1) y pintarla en pantalla.
- Comprobar si las 8 casillas de alrededor son cero.
  Si lo son, ejecutar este mismo algoritmo sobre cada una de las 8 casillas.

La implementación es bastante ilegible (al estar optimizada) y es por eso que os animamos a mejorarla y hacerla más rápida utilizando un algoritmo diferente.

CÓMO PASAR NUESTRO PROGRAMA BASIC A UN FICHERO TAP

Bas2tap es una utilidad muy interesante que permite teclear nuestros programas en BASIC en nuestro ordenador personal y después generar un tap tal y como si lo hubieramos tecleado en el editor del Spectrum.

Si grabamos el Listado 1 como zxmines.bas, podemos generar un tap del juego mediante:

```
bas2tap -a1 -szxmines zxmines.bas
```

La sintaxis (explicada) es:

`-a1` : línea de autoarranque (en qué línea autocomenzará el programa sin necesidad de poner RUN tras el LOAD "").

`-szxmines` : título del programa BASIC.

`zxmines.bas` : programa a "compilar" o "traducir".

Podéis descargar BAS2TAP de la sección de Utilidades de WorldOfSpectrum.

SROMERO

- Fichero basic_zxmines.zip conteniendo: zxmines.*, Makefile, README
- Fichero zxmines.tap (el programa ya compilado)
- Concurso de BASIC 2003 de Bytemaniacos
- SevenuP
- bas2tap



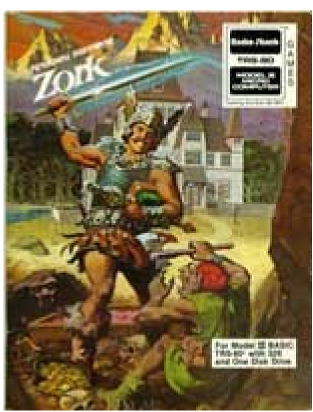
el aventurero

AVENTURAS MULTIPLATAFORMA EN EL +3

El género de las aventuras conversacionales, iniciado con aquella primera aventura llamada precisamente así (Adventure), parece ser actualmente uno de los géneros más antiguos que aún mantienen seguidores (aparte de los seguidores de 'El muro' o 'pong', claro).

Actualmente las aventuras conversacionales suelen generarse, en la escena española, con destino a máquinas virtuales especialmente creadas para las aventuras. Estas máquinas virtuales se caracterizan por poseer intérpretes en múltiples plataformas. Concretando, diremos que hablamos de la máquina Z y la máquina Glulx, dos máquinas virtuales que resultarán totalmente desconocidas para los profanos.

La máquina Z fue creada para ejecutar las aventuras de Infocom, la mas grande creadora de aventuras americana, que la utilizó para sus juegos (con "Zork" como estandarte). La máquina fue diseñada por Joel Berez y Marc Blank en 1979. En 1993, diluida ya Infocom en Activision, la máquina Z se retomó por los aficionados y se creo un lenguaje de programación (Inform, de Graham Nelson) que permitía crear aventuras para esta máquina.



Zork (imagen extraída de http://www.csd.uwo.ca/Infocom/zork.jpg)

La máquina Glulx fue creada años más tarde por Andrew Plotkin para paliar algunas de las limitaciones que suponía la máquina Z, y puede ser programada en Inform y también en Superglús (un PAW ampliado).

Pese a ser la máquina Glulx una máquina más potente, la máquina Z sigue usándose y mucho, fundamentalmente porque si el programador no va a usar las ventajas de Glulx, lo que ocurre muchas veces, es mejor hacerlo para máquina Z. Porque, aunque Glulx es muy portable, teniendo intérpretes en muchas máquinas, la máquina Z lo es más, y lo que para nosotros es más importante, tiene intérprete en Spectrum +3.

¿Qué significa que la máquina Z tiene intérprete en Spectrum +3? Pues que todos los años salen unos cuantos juegos para Spectrum +3 que se nos están escapando, y son aventuras.

Las aventuras hechas con Inform para la máquina Z quedan almacenadas en ejecutables para máquina Z, que vienen a ser una especie de snapshots que contienen los datos y la lógica de la aventura, y que un intérprete puede ejecutar. El intérprete para +3 se llama ZXZVM (ZX Z Virtual Machine), y no sólo lo es para +3, sino para PCW. El intérprete puede cargarse en memoria y cargar una aventura con formato .Z5, que es la extensión que actualmente produce Inform (aunque el intérprete también puede cargar juegos mas viejos en formato .Z4 y .Z3 y el formato con gráficos .Z8, aunque no mostrará los gráficos.

En fin, que gracias a ZXZVM podemos cargar en nuestro +3 unos cuantos juegos de la escena aventurera española, y otros cuantos de la escena aventurera inglesa, y para los muy políglotas pues hasta los de las escenas italiana, francesa o alemana, que las hay.

¿Y cómo? Pues muy fácil, si contamos con una imagen de disco vacía, como las que vienen en las CPDTOOLS y con las utilidades de cpcfs para copiar ficheros del PC a la imagen.

Básicamente necesitamos copiar el cargador BASIC (zxzvm.bas), los dos ficheros que forman el intérprete (ZXIO.BIN y ZXZVM.BIN) y el fichero .Z5, .Z4 o lo que sea en un mismo disco. Tanto el cargador como los dos binarios vienen en el paquete de ZXZVM, mientras que el .Z5 lo tendréis que conseguir por otro lado. Para ello con CPCFS hacemos, sobre una imagen vacía (prueba.dsk), para la aventura prueba.z5:

```
open prueba.dsk
put zxio.bin
put zxzvm.bin
put zxzvm.bas
put prueba.z5
quit
```

Una vez tengamos una imagen de disco de +3 con los cuatro ficheros, debemos abrirla en un +3 o emulador y editar el cargador zxzvm.bas, y cambiar en la línea 40 el nombre del fichero Z5. Mi recomendación es que además salvemos el fichero BAS con el nombre DISK para crear un disco con autoejecución (SAVE "DISK" LINE 10) y luego borréis el .BAS original (ERASE "ZXZVM.BAS"). A partir de ahí basta con seleccionar 'Cargador' o 'Loader' en el menú de vuestro +3 y estaréis jugando a esa aventura.

Para pasarlo a un Spectrum real habrá que utilizar las mismas CPDTools o, si usáis Linux, las libdsk.

No obstante, no iba a ser oro todo lo que reluce, ya que esta posibilidad teórica de jugar aventuras viene a ser perjudicada por tres problemas:

1. Debe cargarse el intérprete en memoria y luego cargar el fichero .Z5, lo cual deja el Z5 más grande a poder cargarse alrededor de los 100k, lo cual descarta muchas aventuras, pero no todas. ZXZVM no tiene control de errores para evitar ocupar mas espacio por lo que el efecto producido es el Spectrum colgado.
2. ZXZVM no es compatible 100% con la máquina Z, y parece colgarse cuando el programador trata de colocar el texto en lugar de escribir todo seguido (para que nos entendamos cuando el autor hizo PRINT AT). Esto no es muy común, pero en algunos juegos pasa sobre todo para el título inicial.
3. Dejo lo peor para el final: el intérprete ZXZVM está hecho en assembler, pero aún así es lento, por lo cual se recomienda jugar en emulador puesto a mayor velocidad de la normal, en lugar de en el +3 real.

El ultimo problema es triste porque relega todo esto al mundo de la emulación (o al del masoquismo). Bueno... quizá alguien quiera probar en un PCW, pero eso tendrá que ser en otro magazine.

A continuación se adjuntan algunos ejemplos de aventuras pasadas a disco +3:

- El libro que se aburría [DSK].
- Una pequeña historia de Navidad [DSK].
- Primera Aventura Experimental y Extraña [DSK].
- La sentencia [DSK].

UTO

- ZXZVM: http://www.ifarchive.org/indexes/if-archiveXinfocomXinterpreters.html
- CPCFS: http://www.426.ch/cpcfs/
- CPDTOOLS: http://www.speccy.org/sinclairmania/arch/utils/cpdtools.zip
- LIBDISK: http://www.seasip.demon.co.uk/Unix/LibDsk
- MÁQUINA Z: http://en.wikipedia.org/wiki/Z-Machine
- LA AVENTURA DE ECSS: http://www.speccy.org/sromero/spectrum/aventura/ecss/
- AVENTURAS EN Z5 Y ESCENA ESPAÑOLA: http://caad.mine.nu
- NEWS INGLESAS: nntp://rec.arts.interactive-fiction
- INFORM: http://caad.mine.nu/informATE
- GLULX: http://www.eblong.com/zarf/glulx/



¡MIS SPRITES SE MUEVEN!

En el artículo anterior aprendimos cómo definir sprites y dibujarlos en pantalla utilizando la librería z88dk sin recurrir a ninguna otra herramienta externa (a menos que quisiéramos dibujar los sprites utilizando algún programa como los mostrados en el número anterior). En esta entrega vamos a ir un poco más allá y vamos a permitir que el usuario pueda mover algún sprite por la pantalla, y que este sprite pueda interaccionar con otros elementos, colisionando con ellos.

¿QUÉ VAMOS A HACER?

Para aprender los conceptos que necesitamos vamos a programar lo que podría ser el código inicial de un juego de carreras de coches, tipo Super Sprint, en los que desde una vista cenital podemos ver el circuito y los distintos participantes. Crearemos un coche utilizando diversos sprites, según hacia donde se esté moviendo el mismo, y crearemos una pista por donde el coche podrá moverse. Evidentemente, esta pista marcará el límite de movimiento de nuestro bólido, por lo que deberemos implementar también algún mecanismo para colisiones, de tal manera que al vehículo le pase algo al contactar con los límites del circuito. Sin más, empecemos.

DEFINIENDO LOS SPRITES

Hasta ahora hemos visto como crear sprites de un tamaño límite de 8x8. Sin embargo, por mucho que lo intentemos, va a ser un poco difícil crear un sprite que represente algo aproximado a un coche con una rejilla tan pequeña. Si nos fijamos en la siguiente imagen, observaremos que tanto para representar al coche en todas las posibles orientaciones, así como los neumáticos que van a formar parte de los límites de la pista, necesitamos utilizar sprites de 10x10. ¿Cómo creamos sprites mayores de 8x8 usando z88dk?

Según vimos anteriormente, un sprite se definía en z88dk como un array de char del tipo:

```
char sprite0[] = { 8, 8, 0x18 ,
0x24 , 0x42 , 0x81 , 0xA5 , 0x81
, 0x7E , 0x00 };
```

donde los dos primeros valores indicaban la altura y la anchura, respectivamente, y los siguientes valores representaban a cada una de las filas del sprite. Si cada columna tenía asignado un valor hexadecimal (estos valores eran, para una anchura de 8, y de derecha a izquierda, los siguientes: 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80), el valor para cada fila era la suma de los valores hexadecimales para los que el pixel correspondiente valía 1.

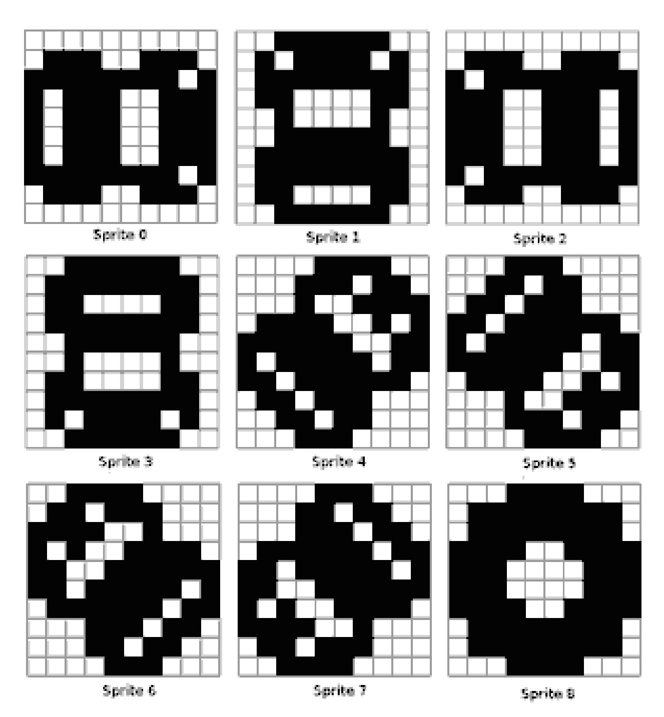


Figura 1: Los sprites que vamos a utilizar en nuestro juego. Representan todas las posibles orientaciones del coche, excepto el último de todos, que representa un neumático. Todos ellos tienen un tamaño de 10x10

Para sprites de más de anchura 8, y hasta una anchura de 16, en ese array de caracteres cada fila se va a representar por un par de números hexadecimales. El primer valor de esa pareja se va a corresponder con los 8 primeros píxeles empezando por la izquierda de la columna, y el segundo con el resto.

En el caso de los sprites que vemos en la figura anterior, donde cada columna tiene una anchura de diez píxeles, cada fila se representaría también por una pareja de valores hexadecimales.

Con la primera pareja codificamos los ocho píxeles de la izquierda, de la misma forma en que lo hacemos con sprites de anchura hasta 8, y con la segunda pareja codificamos los otros dos pixeles que quedan a la derecha (siendo el valor 0x80 el correspondiente al primero y el valor 0x40 el correspondiente al segundo. Si hubiese más de dos, seguiríamos asignando los valores 0x20, 0x10, 0x08, 0x04 y etc. a los siguientes). En la siguiente imagen vemos un ejemplo.

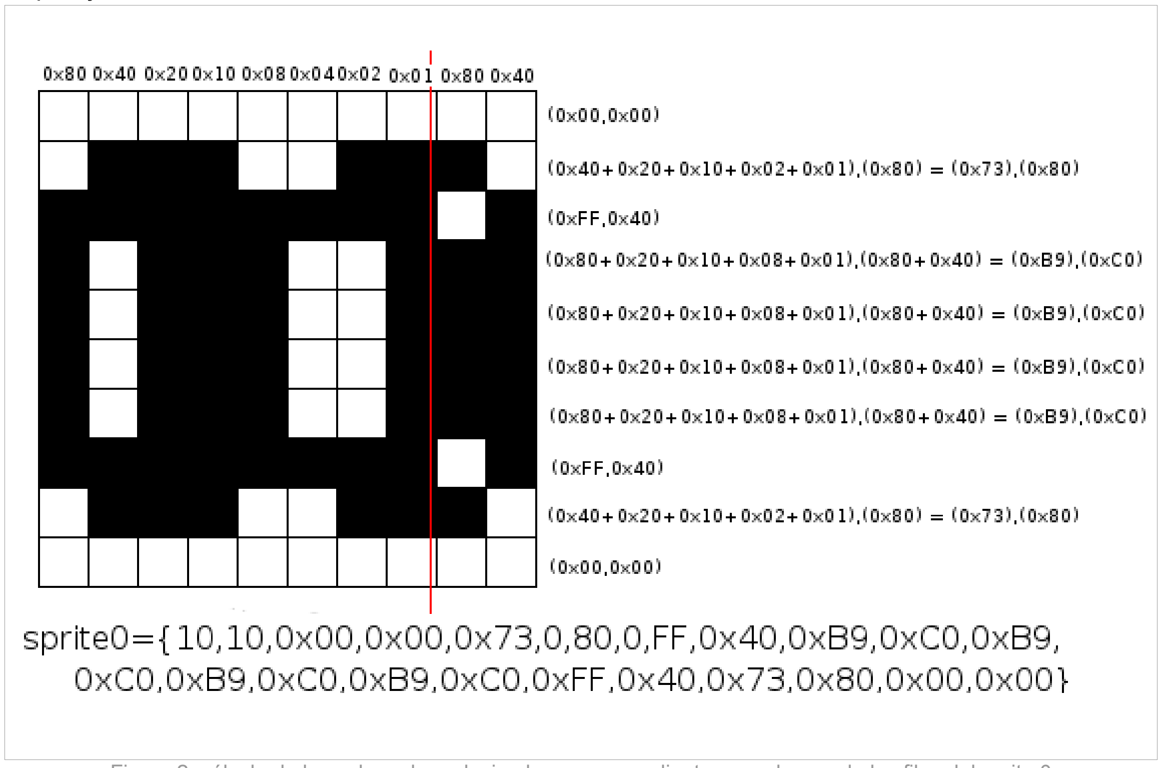


Figura 2: cálculo de los valores hexadecimales correspondientes a cada una de las filas del sprite 0

Podríamos ser malvados y dejar como ejercicio al lector que calcule cómo se codificarían el resto de sprites, pero vamos a ahorrarle el mal trago. A continuación incluímos el código que codifica todos los sprites anteriores, que deberemos introducir dentro de un archivo de cabecera llamado `coches.h`:

```
char sprite0[] = { 10, 10, 0x00 , 0x00 , 0x73 , 0x80 , 0xFF , 0x40 ,
     0xB9 , 0xC0 , 0xB9 , 0xC0 , 0xB9 , 0xC0 , 0xB9 , 0xC0 , 0xFF ,
     0x40 , 0x73 , 0x80 , 0x00 , 0x00  };
char sprite1[] = { 10, 10, 0x3F , 0x00 , 0x5E , 0x80 , 0x7F , 0x80 ,
     0x61 , 0x80 , 0x21 , 0x00 , 0x3F , 0x00 , 0x7F , 0x80 , 0x7F ,
     0x80 , 0x61 , 0x80 , 0x3F , 0x00  };
char sprite2[] = { 10, 10, 0x00 , 0x00 , 0x73 , 0x80 , 0xBF , 0xC0 ,
     0xE7 , 0x40 , 0xE7 , 0x40 , 0xE7 , 0x40 , 0xE7 , 0x40 , 0xBF ,
     0xC0 , 0x73 , 0x80 , 0x00 , 0x00  };
char sprite3[] = { 10, 10, 0x3F , 0x00 , 0x61 , 0x80 , 0x7F , 0x80 ,
     0x7F , 0x80 , 0x3F , 0x00 , 0x21 , 0x00 , 0x61 , 0x80 , 0x7F ,
     0x80 , 0x5E , 0x80 , 0x3F , 0x00  };
char sprite4[] = { 10, 10, 0x0F , 0x00 , 0x1D , 0x80 , 0x13 , 0xC0 ,
     0x79 , 0x40 , 0xFC , 0xC0 , 0xBE , 0xC0 , 0xDF , 0x80 , 0x6E ,
     0x00 , 0x36 , 0x00 , 0x1C , 0x00  };
char sprite5[] = { 10, 10, 0x1C , 0x00 , 0x36 , 0x00 , 0x6E , 0x00 ,
     0xDF , 0x80 , 0xBE , 0xC0 , 0xFC , 0xC0 , 0x79 , 0x40 , 0x13 ,
     0xC0 , 0x1D , 0x80 , 0x0F , 0x00  };
```

```
char sprite6[] = { 10, 10, 0x3C , 0x00 , 0x6E , 0x00 , 0xF2 , 0x00 ,
     0xA7 , 0x80 , 0xCF , 0xC0 , 0xDF , 0x40 , 0x7E , 0xC0 , 0x1D ,
     0x80 , 0x1B , 0x00 , 0x0E , 0x00  };
char sprite7[] = { 10, 10, 0x0E , 0x00 , 0x1B , 0x00 , 0x1D , 0x80 ,
     0x7E , 0xC0 , 0xDF , 0x40 , 0xCF , 0xC0 , 0xA7 , 0x80 , 0xF2 ,
     0x00 , 0x6E , 0x00 , 0x3C , 0x00  };
char sprite8[] = { 10, 10, 0x1E , 0x00 , 0x7F , 0x80 , 0x7F , 0x80 ,
     0xF3 , 0xC0 , 0xE1 , 0xC0 , 0xE1 , 0xC0 , 0xF3 , 0xC0 , 0x7F ,
     0x80 , 0x7F , 0x80 , 0x1E , 0x00  };
```

Y a continuación indicamos el código de un pequeño programa, que llamaremos `coches.c`, que lo único que hará será mostrar por pantalla todos los sprites para comprobar que quedan bonitos. Para ello hacemos uso de la función `putsprite`, cuya sintaxis se explicó en el artículo anterior:

```
#include "games.h"
#include "coches.h"

void main(void)
{
        putsprite(spr_or,1,41,sprite0);
        putsprite(spr_or,21,41,sprite1);
        putsprite(spr_or,41,41,sprite2);
        putsprite(spr_or,61,41,sprite3);
        putsprite(spr_or,1,61,sprite4);
        putsprite(spr_or,21,61,sprite5);
        putsprite(spr_or,41,61,sprite6);
        putsprite(spr_or,61,61,sprite7);
        putsprite(spr_or,1,81,sprite8);
}
```

El archivo de cabecera `games.h` debe ser incluido si queremos utilizar `putsprite`. En la siguiente imagen vemos la salida de este programa.

Figura 3. Los sprites mostrándose en la pantalla de nuestro Spectrum (o de nuestro emulador)

AÑADIENDO MOVIMIENTO

Hacer que un sprite se mueva por pantalla es tan sencillo como borrarlo de donde se encontraba anteriormente y volver a dibujarlo en una nueva posición. Lo interesante será hacer que nuestro coche cambie de aspecto según la orientación en la que se esté moviendo.

Para conseguir esto último, es decir, que se nos muestre un sprite distinto del coche según se esté moviendo hacia arriba, hacia abajo, etc., debemos añadir un par de cambios al archivo `coches.h`. Lo primero que vamos a hacer es almacenar todos los sprites en un único array de arrays de chars (es decir, en un único array de sprites). Esto lo hacemos colocando después de la definición de los diferentes sprites en `coches.h` una línea como la siguiente:

```
char *sprites[9] = { sprite0 , sprite1 , sprite2 , sprite3 , sprite4 ,
      sprite5 , sprite6 , sprite7 , sprite8 };
```

De tal forma que podremos acceder al sprite `i` utilizando, por ejemplo, `sprites[i]` en lugar de `spritei`, lo cual nos va a ser de mucha utilidad. Justo después de esta línea, introducimos las dos

líneas siguientes en `coches.h`:

```
int izquierda[] = {4,6,7,5,1,0,2,3};
int derecha[] = {5,4,6,7,0,3,1,2};
```

Estos dos arrays los utilizaremos para saber, cada vez que giramos, cuál es el sprite que debemos dibujar en la pantalla. Si los sprites están numerados según la primera figura de este artículo, y ahora mismo se nos muestra en la pantalla el sprite 0, si giramos a la izquierda el que se debería mostrar es el 4. Si volvemos a girar a la izquierda, el que debería dibujarse entonces en la pantalla es el 1, etc. Si por el contrario, estamos mostrando el sprite 0 en la pantalla y giramos a la derecha, se nos debería mostrar el sprite 5. Si volvemos a girar a la derecha, se nos debería mostrar el sprite 3, etc.

¿Cómo representamos esto con los arrays indicados anteriormente? Para cada uno de estos dos arrays, la posición `i` representa, para el `sprite i`, cual debería ser el sprite que se debería dibujar si giramos hacia la izquierda, en el caso del primer array, o hacia la derecha, en el caso del segundo. Así, por ejemplo, si estamos mostrando el sprite 1 (el coche hacia arriba) y giramos a la izquierda, el valor para la posición 1 (recordemos que en C la primera posición de los arrays es el 0) en el array izquierda es 6, por lo que deberemos mostrar el sprite 6, correspondiente al coche dirigiéndose en diagonal hacia arriba a la izquierda. Sin embargo, si lo que hacemos es girar hacia la derecha, el valor almacenado en la posición 1 del array derecha es el 4, por lo que deberemos mostrar el sprite 4, correspondiente al coche dirigiéndose hacia arriba a la derecha.

Habiendo hecho estas dos modificaciones, ya podemos incluir aquí el código de un programa que llamaremos `movimiento.c`, que nos pondrá a los mandos de un coche que se mostrará en la pantalla, el cual podremos mover hacia donde queramos, sin ninguna restricción.

```
#include "stdio.h"
#include "ctype.h"
#include "games.h"
#include "coches.h"

void main(void)
{
        int x = 100;
        int y = 100;
        int posicion = 0;

        putsprite(spr_xor,x,y,sprites[0]);

        while(1)
        {
                switch(toupper(getk()))
                {
                       case 'O':
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                posicion = izquierda[posicion];
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                break;
                        case 'P':
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                posicion = derecha[posicion];
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                break;
                }
        }
}
```

De momento vamos a comenzar con los giros, y ya surcaremos después la pantalla a toda velocidad. El código anterior nos dibuja el coche en una zona cercana al centro de la pantalla, y nos permite girarlo hacia la izquierda y hacia la derecha empleando las teclas o y p respectivamente. Explicamos el código y más adelante indicamos un problema que se muestra durante la ejecución y cómo solucionarlo.

Lo primero que hacemos es incluir algunos archivos de cabecera correspondientes a la librería z88dk para poder hacer uso de algunas funcionalidades de la librería. El segundo de ellos, `ctype.h` permite utilizar la función `toupper`, que devuelve la cadena que se le pasa como parámetro pasada a mayúsculas. El tercero que incluimos, `games.h`, es el que hemos utilizado hasta ahora para poder utilizar `putsprite`, que ya sabemos de sobra lo que hace. Y finalmente, el primero de ellos, `stdio.h`, lo incluimos para poder leer de teclado con la función `getk`. Evidentemente, también debemos incluir `coches.h`, pues es donde se definen nuestros sprites, el array que los contiene a todos ellos, y los arrays `izquierda` y `derecha` cuyo funcionamiento hemos explicado anteriormente.

Una vez que comienza el programa principal, empezamos a definir variables. En el caso de x e y, su cometido es tan simple como indicarnos en que posición de la pantalla se va a encontrar el coche. Ahora que vamos a tener el coche fijo girando a lo mejor no parece de utilidad, pero luego, más tarde, cuando lo movamos, sí que va a ser así. La última variable que definimos es `posicion` encargada de indicarnos que sprite estamos dibujando en pantalla. Si el array `sprites` contiene todos los sprites definidos, al hacer un `putsprite` de `sprites[posicion]`, lo que estaremos haciendo es dibujar por pantalla el sprite almacenado en la posición posicion de dicho array. En nuestro caso, esta variable comienza valiendo 0, por lo que cuando ejecutemos el programa, veremos que el coche, nada más empezar, se encuentra preparado para correr orientado hacia la derecha.

Y una vez dibujamos el coche por primera vez, utilizando `putsprite`, ya estamos preparados para entrar en el bucle principal. Es muy importante fijarse que para el modo de dibujado (primer parámetro de `putsprite`) estamos utilizando `spr_xor`, correspondiente a la operación lógica OR EXCLUSIVA. Usamos este modo para poder borrar sprites de una forma muy cómoda. De forma intuitiva, lo único que debemos saber es que si dibujamos un sprite usando or exclusiva en una zona de la pantalla que esté vacía, el sprite se dibujará tal cual en la pantalla. Si dibujamos el sprite usando or exclusiva en una zona de la pantalla donde ese sprite ya estuviera dibujado, se borrará sin modificar el resto de los píxeles de por alrededor, solo se borrarán los píxeles que formen parte del coche.

Por lo tanto, para mover, lo que haremos será dibujar el sprite usando or exclusiva en la misma posición en la que se encontrara anteriormente, y después vovler a dibujar usando or exclusiva en su nueva posición.

El bucle principal se va a ejecutar de forma indefinida (`while (1)`). En el mismo, leeremos constantemente lo que el usuario introduzca por el teclado utilizando `getk`. Hemos de tener en cuenta dos cosas: la función `getk` es no bloqueante, lo cual quiere decir que si el usuario no pulsa ninguna tecla, el programa no se queda detenido en esa instrucción, y utilizamos `getk` combinado con `toupper`, que transforma lo que se le pasa como parámetro a mayúsculas, para que el movimiento funcione igual tanto si el usuario pulsa las teclas con el bloqueo mayúsculas activado o sin el. Por eso, dentro del switch, a la hora de comprobar que tecla se ha pulsado, comparamos la entrada con 'O' y 'P' en lugar de con 'o' y 'p'.

Si la tecla que se ha pulsado es la 'o', deseamos girar hacia la izquierda. Dibujamos el coche en la posición actual usando or exclusiva, de tal forma que se borra. Con la línea `posicion = izquierda[posicion]` lo que hacemos es averiguar que sprite es el que tenemos que dibujar a continuación, según la orientación en la que nos encontremos. Finalmente volvemos a dibujar el coche con las mismas coordenadas (x,y), pero utilizando un sprite distinto, correspondiente a haber girado el coche a la izquierda. En el caso de que se hubiera pulsado la tecla 'p' se realizaría la misma operación, pero utilizando el array `derecha`.

Ahora ya podemos compilar y ejecutar y comprobaremos como podemos girar nuestro coche pulsando las teclas 'o' y 'p'. Sin embargo, veremos que se produce un efecto extraño. Al mantener una tecla pulsada durante un tiempo, es como si estuviéramos escribiendo en BASIC: se gira una vez, y tras un breve tiempo, ya se gira sin interrupción. Más tarde averiguaremos como resolver este pequeño contratiempo. Antes, veamos como podemos hacer que nuestro coche acelere y frene.

Queremos que pulsando la tecla 'q' el coche vaya acelerando hasta una velocidad máxima (y por lo tanto, que se vaya moviendo conforme a esa velocidad) y que al pulsar la tecla 'a', el coche vaya frenando hasta detenerse. A continuación vemos como lo hemos resuelto, en el siguiente programa, que sustituirá a nuestro anterior `movimiento.c` (el código en rojo es nuevo):

```
#include "stdio.h"
#include "ctype.h"
#include "games.h"
#include "coches.h"

#define CICLOS_BASE 300;

void main(void)
{
        int x = 100;
        int y = 100;
        int posicion = 0;

        int velocidad = 0;
        int ciclos = CICLOS_BASE;

        putsprite(spr_xor,x,y,sprites[0]);

        while(1)
        {
                switch(toupper(getk()))
                {
                        case 'Q':
                                if (velocidad < 50)
                                {
                                        velocidad = velocidad + 5;
                                }
                                break;
                        case 'A':
                                if (velocidad > 0)
                                {
                                        velocidad = velocidad - 5;
                               }
                                break;
                        case 'O':
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                posicion = izquierda[posicion];
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                break;
                        case 'P':
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                posicion = derecha[posicion];
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                break;
                }

                if (velocidad > 0)
                {
                        ciclos = ciclos - velocidad;
                        if (ciclos < 0)
                        {
                                ciclos = CICLOS_BASE;
                                putsprite(spr_xor,x,y,sprites[posicion]);
                                switch(posicion)
                                {
                                        case 0:
                                                x = x + 1;
                                                break;
                                        case 1:
                                y = y - 1;
                                                break;
                                        case 2:
```

```
                                        x = x - 1;
                                        break;
                                case 3:
                                        y = y + 1;
                                        break;
                                case 4:
                                        x = x + 1;
                                        y = y - 1;
                                      break;
                                case 5:
                                        x = x + 1;
                                        y = y + 1;
                                        break;
                                case 6:
                                        x = x - 1;
                                        y = y - 1;
                                        break;
                                case 7:
                                        x = x - 1;
                                        y = y + 1;
                                        break;
                        }
                        putsprite(spr_xor,x,y,sprites[posicion]);
                }
            }
        }
}
```

Lo más evidente es que necesitaremos una variable, en este caso llamada `velocidad`, que nos permita almacenar la velocidad del coche en un momento dado. Es importante tener en cuenta que en un principio el coche estará parado, por lo que esta velocidad valdrá cero. Hemos añadido también código que hará que nuestra velocidad aumente o disminuya, hasta una velocidad máxima o mínima, al pulsar las teclas 'q' y 'a', respectivamente (este código es el que se ha añadido dentro del switch).

Lo que a lo mejor queda un poco más esotérico es ver cómo hacemos que el coche, una vez que adquiere velocidad, pueda moverse. La parte del código encargada de esto es la que se encuentra al final. Este código, como es obvio, solo se ejecutará si el coche tiene velocidad. El coche se moverá cuando una variable, llamada `ciclos`, y a la que se le va restando el valor de la velocidad en cada iteración del bucle principal, llegue a cero. Evidentemente, cuanta mayor sea la velocidad, más rápido llegará ciclos a 0 y cada menos iteraciones del bucle principal se moverá el coche.

Si los ciclos llegan a cero, nos disponemos a mover el coche. Lo primero es volver a hacer que ciclos valga su valor inicial, para volver a comenzar el proceso al terminar de mover. Lo siguiente es borrar el coche de su posición anterior, dibujándolo en dicha posición usando el modo or exclusiva. Y después, con un switch, cambiamos el valor de la coordenada x y/o y según la orientación del coche, dada por la variable `posicion`. Así, por ejemplo, si `posicion` vale 0, eso quiere decir que el coche está orientado hacia la derecha, por lo que aumentamos el valor de la coordenada x. Si `posicion` valiera 4, el coche estaría orientado hacia arriba a la derecha, por lo que disminuiríamos el valor de y y aumentaríamos el de x. Y así con el total de las 8 orientaciones. Finalmente, dibujamos el coche en su nueva posición x e y.

Aquí encontramos varias ventajas con respecto al BASIC. Primero, que los gráficos definidos por el usuario (en este caso, sprites) no tienen por qué estar limitados a un tamaño de 8 por 8, y segundo, que estos sprites pueden moverse utilizando incrementos que sean cualquier múltiplo de un pixel, mientras que en basic debemos mover los UDGs de 8 en 8.

Al ejecutar este programa, veremos como podemos acelerar y frenar, y el coche se moverá, sin ninguna restricción. Si el coche sale de la pantalla, observaremos como este entra por el extremo opuesto y el programa seguirá funcionando sin problemas. Sin embargo, se sigue produciendo el mismo efecto que comentábamos antes, el teclado parece comportarse como si estuviéramos en BASIC... vamos a solucionarlo dándole valor a dos variables del sistema, ¡pero que no se asuste nadie!.

Las variables del sistema se podrían entender como determinadas direcciones de memoría que modifican el comportamiento del sistema según su valor (más información en la Microhobby especial nº2). En nuestro caso concreto, las variables del

sistema que vamos a modificar son REPDEL y REPPER, correspondientes a las direcciones de memoria 23561 y 23562. La primera de ellas indica el tiempo en cincuentavos de segundo que se debe tener pulsada una tecla para que esta se comience a repetir, y la segunda indica el tiempo en cincuentavos de segundo que tarda en producirse esta repetición una vez que se comienza. Si le damos un valor de 1 a estas variables, conseguiremos una respuesta del teclado rápida, y nos desharemos del efecto tan fastidioso que hemos comentado antes.

En el siguiente código se muestran, en rojo, los cambios que deberíamos realizar al comienzo del programa `movimiento.c`:

```
#include "stdio.h"
#include "ctype.h"
#include "games.h"
#include "coches.h"

#define CICLOS_BASE 300;

void main(void)
{
        int x = 100;
        int y = 100;
        int posicion = 0;

        char *puntero1 = (char *) 23561;
        char *puntero2 = (char *) 23562;


        int velocidad = 0;
        int ciclos = CICLOS_BASE;

        putsprite(spr_xor,x,y,sprites[0]);

        while(1)
        {
                *puntero1 = 1;
                *puntero2 = 1;
                switch(toupper(getk()))
                {
```

Al mejorar la respuesta del teclado nos ha surgido un nuevo problema... ¡el coche gira demasiado rápido!

Será conveniente añadir un contador para que el coche no gire nada más pulsar la tecla correspondiente; mejor que gire cuando la tecla lleve un rato pulsada. A continuación mostramos, en rojo, el código que deberíamos añadir:

```
        int girando = 0;
        int contador_izquierda = 0;
        int contador_derecha = 0;
        int velocidad = 0;
        int ciclos = CICLOS_BASE;

        putsprite(spr_xor,x,y,sprites[0]);

        while(1)
        {
                *puntero1 = 1;
                *puntero2 = 1;

        girando = 0;

                switch(toupper(getk()))
                {
```

```
                case 'Q':
                        if (velocidad < 50)
                        {
                                velocidad = velocidad + 5;
                        }
                        break;
                case 'A':
                        if (velocidad > 0)
                        {
                                velocidad = velocidad - 5;
                        }
                        break;
                case 'O':
                        contador_derecha = 0;
                        contador_izquierda =
                        contador_izquierda + 1;
                        girando = 1;
                        if (contador_izquierda == 3)
                        {
                                putsprite(spr_xor,x,y,sprites
                                [posicion]);
                                posicion = izquierda[posicion];
                                putsprite(spr_xor,x,y,sprites
                                [posicion]);
                                contador_izquierda = 0;
                        }
                        break;
               case 'P':
                        contador_izquierda = 0;
                        contador_derecha = contador_derecha + 1;
                        girando = 1;
                        if (contador_derecha == 3)
                        {
                                putsprite(spr_xor,x,y,sprites
                                [posicion]);
                                posicion = derecha[posicion];
                                putsprite(spr_xor,x,y,sprites
                                [posicion]);
                                contador_derecha = 0;
                         }
                         break;
        }

        if (girando == 0)
        {
                contador_izquierda = 0;
                contador_derecha = 0;
        }


        if (velocidad > 0)
        {
```

Nos vamos a basar en tres nuevas variables, girando, contador_izquierda y contador_derecha. Las dos últimas son las que nos van a indicar cuándo hacer el giro. Cada vez que le demos a la tecla de giro a la izquierda, aumentará el valor de contador_izquierda, y cada vez que le demos a la tecla de giro a la derecha, aumentará el valor de contador_derecha. Cuando alguna de ellas valga 3, giraremos a la izquierda o a la derecha, respectivamente.

Una medida que tomamos es que al girar hacia la izquierda, ponemos a cero la variable contador_derecha, que nos dice cuanto tiempo hemos estado girando a la derecha, y viceversa, cuando giramos a la derecha, ponemos a cero la variable contador_izquierda, que nos dice cuánto hemos girado hasta la izquierda hasta el momento. Así evitamos que, en el caso de haber estado girando hacia la izquierda durante un

tiempo, por ejemplo, empecemos a girar a la derecha, y que tan solo haga falta rozar la tecla de giro izquierda para volver a girar a la izquierda, lo cual no es demasiado real.

Otra medida que tomamos está relacionada con la variable `girando`. Si dejamos de pulsar las teclas de giro, no es demasiado real que si volvemos a pulsarlas otra vez pasado un rato y durante muy poco tiempo, giremos. Para solucionar esto, lo que hacemos simplemente es: darle a la variable `girando` el valor 0 cada vez que entramos en el bucle, darle el valor 1 en el caso de que pulsemos giro izquierda o giro derecha, y por último, poner `contador_izquierda` y `contador_derecha` a cero en el caso de que no se haya girado en esa iteración, y por lo tanto, en el caso de que `girando` valga cero... es bastante sencillo de comprender.

¿Y ya está? ¡No! Más problemas se interponen entre nosotros y un sprite en movimiento. Si ejecutamos el código anterior veremos como el coche... ¡no gira! ¿Es que hemos hecho algo mal? No, la algoritmia está bien, pero tenemos un problema de interrupciones. El programa no espera a que pulsemos una tecla, por lo que a lo mejor nosotros estamos pulsando la tecla P todo el rato, pero el programa no llega a leer el teclado a tiempo y `girando` vuelve a valer 0.

Vamos a hacer que el bucle principal solo avance cuando se produzca una interrupción. Dichas interrupciones se producirán en dos casos, cuando pulsemos una tecla (lo cual es lo que queremos) y cuando vaya a comenzar el refresco de la pantalla (cuando el haz de electrones se sitúe de nuevo en el punto 0,0 de la pantalla para empezar a dibujar, de izquierda a derecha y de arriba a abajo). Para ello tendremos que utilizar... ¡ensamblador!. Pero no hay que preocuparse, porque lo solucionaremos todo con una línea de código.

La instrucción en ensamblador que detiene la ejecución hasta que se produce una interrupción es HALT. Y en z88dk, ejecutar una instrucción en ensamblador en cualquier momento es tan sencillo como hacer uso de la función `asm`, que recibe como parámetro una cadena conteniendo la instrucción que deseamos ejecutar. Por lo tanto, resolver de una vez por todas nuestros problemas de movimiento es tan simple como colocar la instrucción `asm("HALT")` en los lugares adecuados. A continuación se muestra todo el código de `movimiento.c` tal como quedaría después de los cambios (marcados en rojo).

```
#include "stdio.h"
#include "ctype.h"
#include "games.h"
#include "coches.h"

#define CICLOS_BASE 20;

void main(void)
{
    int x = 100;
    int y = 100;
    int posicion = 0;

    char *puntero1 = (char *) 23561;
    char *puntero2 = (char *) 23562;


    int girando = 0;
    int contador_izquierda = 0;
    int contador_derecha = 0;
    int velocidad = 0;
    int ciclos = CICLOS_BASE;

    putsprite(spr_xor,x,y,sprites[0]);

    while(1)
    {
        *puntero1 = 1;
        *puntero2 = 1;
        asm("halt");

        girando = 0;
```

```
        switch(toupper(getk()))
        {
            case 'Q':
                if (velocidad < 30)
                {
                    velocidad = velocidad + 1;
                }
                break;
            case 'A':
                if (velocidad > 0)
                {
                    velocidad = velocidad - 1;
                }
                break;
            case 'O':
                contador_derecha = 0;
                contador_izquierda = contador_izquierda + 1;
                girando = 1;
                if (contador_izquierda == 3)
                {
                    asm("halt");
                    putsprite(spr_xor,x,y,sprites[posicion]);
                    posicion = izquierda[posicion];
                    putsprite(spr_xor,x,y,sprites[posicion]);
                    contador_izquierda = 0;
                }
                break;
            case 'P':
                contador_izquierda = 0;
                                contador_derecha = contador_derecha + 1;
                                girando = 1;
                                if (contador_derecha == 3)
                                {
                    asm("halt");
                                    putsprite(spr_xor,x,y,sprites[posicion]);
                                    posicion = derecha[posicion];
                                    putsprite(spr_xor,x,y,sprites[posicion]);
                                        contador_derecha = 0;
                                 }
                                 break;
        }

                if (girando == 0)
            {
                contador_izquierda = 0;
                contador_derecha = 0;
        }


        if (velocidad > 0)
        {
            ciclos = ciclos - velocidad;
            if (ciclos < 0)
            {
                ciclos = CICLOS_BASE;
                asm("halt");
                putsprite(spr_xor,x,y,sprites[posicion]);
                switch(posicion)
                {
                    case 0:
                        x = x + 1;
                        break;
                    case 1:
                y = y - 1;
```

```
                        break;
                    case 2:
                        x = x - 1;
                        break;
                    case 3:
                        y = y + 1;
                        break;
                    case 4:
                        x = x + 1;
                        y = y - 1;
                        break;
                    case 5:
                        x = x + 1;
                        y = y + 1;
                        break;
                    case 6:
                        x = x - 1;
                        y = y - 1;
                        break;
                    case 7:
                        x = x - 1;
                        y = y + 1;
                        break;
                }
                putsprite(spr_xor,x,y,sprites[posicion]);
            }
        }
    }
}
```

Como se puede comprobar, se ha incluido un `halt` antes de la lectura de teclado, para detener el programa hasta que se pulse una tecla, y después se ha incluido un `halt` antes de borrar y volver a dibujar el sprite, para esperar el refresco de la pantalla. Como consecuencia, el coche irá más lento, es por ello que también hemos modificado el valor de `CICLOS_BASE`, la velocidad máxima, y el incremento y decremento de la velocidad. ¿Y os habéis fijado? Gracias a que hemos sincronizado el movimiento del coche con el refresco de la pantalla... ¡éste ha dejado de parpadear al moverse!

Por último, una cosa curiosa: si mientras movemos el coche pasamos por encima de donde pone `Bytes: movimiento` (correspondiente a la carga desde cinta) veremos como el borrado y la escritura del sprite del coche no afecta en lo mas mínimo a los píxeles que forman parte de ese texto; esto es sin duda otra de las grandes ventajas de usar el modo de dibujado de sprites or exclusiva.

Y AHORA, PONGAMOS EL COCHE EN UNA PISTA

Por último vamos a dibujar una pista de neumáticos para limitar un poco el movimiento de nuestro bólido. Vamos además a hacerlo de tal forma que sea muy fácil modificar el trazado, y así la diversión se multiplique durante meses. Si recordamos, el último de los sprites que habíamos definido era el correspondiente a un neumático. Este sprite, como todos los anteriores, tiene un tamaño de 10x10 (quizá un poco grande, para hacer circuitos más complicados quizás no hubiera estado mal utilizar sprites de neumáticos más pequeños).

Dentro de `coches.h` es donde definiremos el trazado de la pista. Si consideramos la pantalla como un array de 18x24 casillas de tamaño 10x10 (el mismo que los neumáticos), podemos indicar el recorrido de la pista creando un array, donde colocaremos ceros en aquellas casillas donde no vaya a haber neumático, y unos en las casillas en las que sí. Al principio de `coches.h` definimos unas constantes para el tamaño del circuito:

```
#define ALTURA_CIRCUITO 18
#define ANCHURA_CIRCUITO 24
```

y al final del mismo archivo incluimos el siguiente array:

```
short circuito1[] =  { 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    0, 0, 0, 0, 0, 0, 0 };
short circuito2[] =  { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    1, 1, 1, 0, 0, 0, 0 };
short circuito3[] =  { 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 1, 1, 0, 0 };
short circuito4[] =  { 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 1, 0 };
short circuito5[] =  { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 1 };
short circuito6[] =  { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 1 };
short circuito7[] =  { 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
    0, 0, 0, 0, 0, 0, 1 };
short circuito8[] =  { 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,
    0, 0, 0, 0, 0, 0, 1 };
short circuito9[] =  { 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,
    0, 0, 0, 0, 0, 0, 1 };
short circuito10[] = { 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 0, 0, 1, 0 };
short circuito11[] = { 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 0, 1, 0, 0 };
short circuito12[] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 1, 0, 0 };
short circuito13[] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 1, 0, 0, 0 };
short circuito14[] = { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 1, 0, 0, 0 };
short circuito15[] = { 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 0, 0, 0, 0 };
short circuito16[] = { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
    1, 1, 0, 0, 0, 0, 0 };
short circuito17[] = { 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,
    0, 0, 0, 0, 0, 0, 0 };
short circuito18[] = { 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0 };

short int *circuito[ALTURA_CIRCUITO] = {circuito1, circuito2, circuito3,
    circuito4, circuito5, circuito6, circuito7, circuito8, circuito9,
    circuito10, circuito11, circuito12, circuito13, circuito14, circuito15,
    circuito16, circuito17, circuito18};
```

Con un poco de imaginación podemos vislumbrar en ese array un circuito cerrado en forma de O. Si ahora creamos un fichero llamado `juego.c`, que contenga el mismo código que `movimiento.c`, pero añadiéndole el que aperece en color rojo a continuación, podremos por fin ver nuestro circuito en la pantalla, tal como se ve en la siguiente imagen:

```
void main(void)
{
        int x = 100;
        int y = 140;
        int posicion = 0;
        short int i;
        short int j;
```

```
char *puntero1 = (char *) 23561;
char *puntero2 = (char *) 23562;

int girando = 0;
int contador_izquierda = 0;
int contador_derecha = 0;
int velocidad = 0;
int ciclos = CICLOS_BASE;

for (i=0;i<30;i++)
        printf("\n");

for (i=0;i

putsprite(spr_xor,x,y,sprites[0]);

while(1)
{
        *puntero1 = 1;
        *puntero2 = 1;
        asm("halt");
```

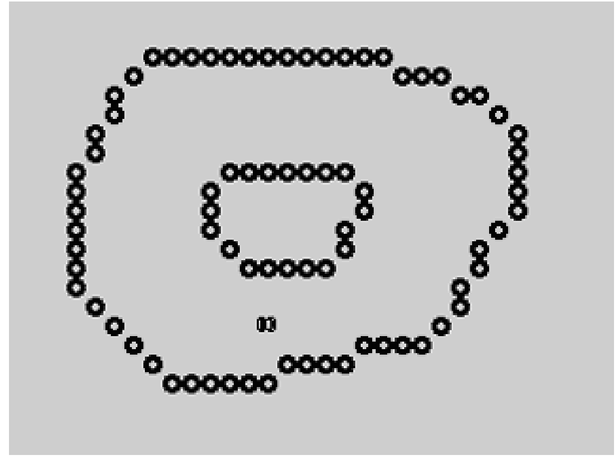


Figura 4. Nuestro coche dispuesto a ser el rey de la pista

Hay que tener en cuenta que hemos cambiado el valor inicial de la coordenada y del coche para que éste quede dentro de la pista. También hemos creado dos variables, `i` y `j` que nos van a servir de contadores en un par de bucles.

Este par de bucles son los dos que se muestran en rojo. En el primero lo único que hacemos es escribir en la pantalla 30 líneas en blanco para borrarla (es por eso que en la captura anterior ya no se ve lo de `Bytes: juego.tap` que molestaba tanto). Es en el segundo en el que dibujamos el circuito.

Si hemos decidido anteriormente que el array `circuito` nos iba a indicar si en cada celda de 10x10 píxeles en las que dividíamos la pantalla había o no un neumático, lo que haremos para dibujar la pista no es más que recorrer todas las posiciones del array y dibujar en la pantalla un neumático en la posición `i*10+1`, `j*10+1`, siendo `i` y `j` las coordenadas dentro del array donde hemos leído un 1.

De acuerdo, ya tenemos un coche moviéndose y un circuito... pero cuando el coche llega hasta alguno de los neumáticos, en lugar de producirse una espectacular explosión, lo atraviesa como si nada y sigue su camino. Tenemos, sin duda, que añadir colisiones.

Para detectar cuando nuestro coche colisiona con uno de los neumáticos que forman parte de los límites de la pista, deberemos añadir una condición `if` antes de cada desplazamiento del mismo, de tal forma que si ese desplazamiento va a provocar que coche y neumático entren en contacto, el coche se pare totalmente (dada la poca velocidad a la que se ha programado el coche en nuestro programa, y a nuestra incapacidad de momento para crear grandes explosiones usando z88dk en el Spectrum, vamos a suponer que al contactar con un neumático el coche solamente queda parado).

A continuación mostramos lo que sería el código completo de `juego.c`, con las partes nuevas en color rojo, como siempre:

```
#include "stdio.h"
#include "ctype.h"
#include "games.h"
#include "coches.h"

#define CICLOS_BASE 20;

void main(void)
{
    int x = 100;
    int y = 140;
    int posicion = 0;
    short int i;
    short int j;

    char *puntero1 = (char *) 23561;
    char *puntero2 = (char *) 23562;

    int girando = 0;
    int contador_izquierda = 0;
    int contador_derecha = 0;
    int velocidad = 0;
    int ciclos = CICLOS_BASE;

    for (i=0;i<30;i++)
        printf("\n");

    for (i=0;i< 30)
                {
                    velocidad = velocidad + 1;
                }
                break;
            case 'A':
                if (velocidad > 0)
                {
                    velocidad = velocidad - 1;
                }
                break;
            case 'O':
                contador_derecha = 0;
                contador_izquierda = contador_izquierda + 1;
                girando = 1;
                if (contador_izquierda == 3)
                {
                    asm("halt");
                    putsprite(spr_xor,x,y,sprites[posicion]);
                    posicion = izquierda[posicion];
                    putsprite(spr_xor,x,y,sprites[posicion]);
                    contador_izquierda = 0;
                }
                break;
            case 'P':
                contador_izquierda = 0;
                contador_derecha = contador_derecha + 1;
                girando = 1;
                if (contador_derecha == 3)
                {
                    asm("halt");
                    putsprite(spr_xor,x,y,sprites[posicion]);
                    posicion = derecha[posicion];
                    putsprite(spr_xor,x,y,sprites[posicion]);
                      contador_derecha = 0;
                }
```

```
            break;
    }

    if (girando == 0)
    {
        contador_izquierda = 0;
        contador_derecha = 0;
    }

    if (velocidad > 0)
    {
        ciclos = ciclos - velocidad;
        if (ciclos < 0)
        {
            ciclos = CICLOS_BASE;
            asm("halt");
            putsprite(spr_xor,x,y,sprites[posicion]);
            switch(posicion)
            {
                case 0:

                    if (circuito[y/10][(x + 9)/10] == 1 ||
                        (y%10 != 0 && circuito[(y+7)/10][(x+9)/10] == 1))

                        velocidad = 0;
                    else
                        x = x + 1;
                    break;
                case 1:

                    if (circuito[(y-2)/10][x/10] == 1 ||
                        (x%10 != 0 && circuito[(y-2)/10][(x+7)/10] == 1))

                        velocidad = 0;
                    else
                        y = y - 1;
                    break;
                case 2:

                    if (circuito[y/10][(x-2)/10] == 1 ||
                        (y%10 != 0 && circuito[(y+7)/10][(x-2)/10] == 1))

                        velocidad = 0;
                    else
                        x = x - 1;
                    break;
                case 3:

                    if (circuito[(y+9)/10][x/10] == 1 ||
                        (x%10 != 0 && circuito[(y+9)/10][(x+7)/10] == 1))

                        velocidad = 0;
                    else
                        y = y + 1;
                    break;
                case 4:

                    if (circuito[y/10][(x + 9)/10] == 1 ||
                        (y%10 != 0 && circuito[(y+7)/10][(x+9)/10] == 1) ||
                        circuito[(y-2)/10][x/10] == 1 ||

                        (x%10 != 0 && circuito[(y-2)/10][(x+7)/10] == 1))

                        velocidad = 0;
```

```
                        else
                        {
                            x = x + 1;
                            y = y - 1;
                        }
                        break;
                    case 5:

                        if (circuito[y/10][(x + 9)/10] == 1 ||
                            (y%10 != 0 && circuito[(y+7)/10][(x+9)/10] == 1) ||
                            circuito[(y+9)/10][x/10] == 1 ||
                            (x%10 != 0 && circuito[(y+9)/10][(x+7)/10] == 1))

                            velocidad = 0;
                        else
                        {
                            x = x + 1;
                            y = y + 1;
                        }
                        break;
                    case 6:

                        if (circuito[y/10][(x-2)/10] == 1 ||
                            (y%10 != 0 && circuito[(y+7)/10][(x-2)/10] == 1) ||
                            circuito[(y-2)/10][x/10] == 1 ||
                            (x%10 != 0 && circuito[(y-2)/10][(x+7)/10] == 1))

                            velocidad = 0;
                        else
                        {
                            x = x - 1;
                            y = y - 1;
                        }
                        break;
                    case 7:

                        if (circuito[y/10][(x-2)/10] == 1 ||
                            (y%10 != 0 && circuito[(y+7)/10][(x-2)/10] == 1) ||
                            circuito[(y+9)/10][x/10] == 1 ||
                            (x%10 != 0 && circuito[(y+9)/10][(x+7)/10] == 1))

                            velocidad = 0;
                        else
                        {
                            x = x - 1;
                            y = y + 1;
                        }
                        break;
                }
                putsprite(spr_xor,x,y,sprites[posicion]);
            }
        }
    }
}
```

Simplemente, antes de mover el coche, en la última parte del bucle principal, comprobamos si va a haber una colisión con alguno de los neumáticos. En el caso de que sea así ponemos la velocidad a cero, y en caso contrario realizamos el movimiento de forma normal. En todas las comprobaciones miramos a ver si en la posición del array circuito correspondiente a donde estaría mas o menos situado el coche (recordemos que en circuito cada posición se corresponde con 10x10 píxeles de la pantalla) hay un 1.

Analicemos, por ejemplo, para el `case 0`, que se corresponde con el coche moviéndose hacia la derecha, y el resto de condiciones se podrá sacar de la misma forma (hemos de aclarar que en los cuatro últimos `case`, al tratarse de movimientos diagonales, la comprobación debe ser en los dos sentidos, eje `x` y eje `y`):

```
if (circuito[y/10][(x + 9)/10] == 1 ||
     (y%10 != 0 && circuito[(y+7)/10][(x+9)/10] == 1))
```

Básicamente la comprobación se compone de dos partes, la que está a la izquierda del OR (que en C, recordemos, se codifica con ||) y la que está a la derecha. Si en el array `circuito` decíamos que cada posición se correspondía con una región de 10x10 píxeles de la pantalla, cuando el coche está en las coordenadas `(y,x)`, tendríamos que comprobar en la posición `(y/10,x/10)` del array `circuito` (en C, las divisiones enteras se redondean hacia abajo).

La anchura y la altura del coche es 10, por lo tanto, si queremos movernos hacia la derecha, aumentando el valor de x en 1, debemos comprobar si el pixel situado en el extremo derecho, es decir, el pixel situado en `x+11`, estará en contacto con algún pixel de algún neumático. Esto es equivalente a comprobar si en la posición `[y/10][(x+11)/10]` de `circuito` hay almacenado un 1. Si es así habrá colisión, por lo que no nos interesará avanzar, se cumplirá la condición, y `velocidad` pasará a valer 0. Lo que ocurre es que en nuestro caso, en lugar de `x+11` hemos usado `x+9` ya que, por ensayo y error hemos comprobado que con este valor el coche solo colisionara cuando este totalmente pegado al neumático (nuestros sprites son de 10x10, pero algunos de ellos no ocupan toda la anchura o toda la altura).

Esto estaría bien si el coche solo pudiera moverse en el eje `y` en múltiplos de 10, pero esto no siempre es así. Si el coche estuviera en `(12,12)` (valiendo 12 la coordenada `y`, por lo tanto), al movernos a la derecha, podríamos colisionar con un neumático almacenado en el array `circuito` en (1,2), pero también con el que estuviera en (2,2) (el de abajo). Por lo tanto, la segunda condición comprueba si, en el caso de que la coordenada y no tenga un valor múltiplo de 10 (es decir, cuando el resultado de la operación resto, especifacada en C con % y que calcula el resto de la división entera, devuelva un valor distinto de cero), habría colisión con un neumático situado abajo a la derecha. Como en el caso de la `x`, en lugar de sumar 11 a `y`, añadimos un valor calculado a ojo que nos asegura que el coche quedaría pegadito en el caso de una colisión vertical.

Se podrían cumplir las dos condiciones a la vez, pero nos es indiferente, con que se cumpla una de las dos ya hay colisión.

En el caso de movernos hacia la izquierda, en lugar de sumar a `x` restamos, y en el caso de movernos hacia arriba o hacia abajo, hacemos lo mismo pero cambiando lo que sumamos o restamos a `x` por lo que sumamos o restamos a `y` y viceversa.

¿Y eso es todo? Bueno... hemos de ser sinceros y admitir que debido a la inclusión del código que detecta las colisiones, ha vuelto el parpadeo del coche al moverse, sobre todo por la parte superior de la pantalla. No le da tiempo al programa a sincronizar el movimiento con el refresco de la pantalla, debido a lo complejo de los cálculos. Esto, a grandes rasgos, se puede mejorar, aunque solo sea un poco, creando dos nuevas variables, `x_anterior` e `y_anterior`, y dejando el código de detección de colisiones de la siguiente manera (calculando las colisiones antes de esperar el refresco de la pantalla y borrar y dibujar el sprite):

```
        if (velocidad > 0)
        {
            ciclos = ciclos - velocidad;
            if (ciclos < 0)
            {
                ciclos = CICLOS_BASE;
                y_anterior = y;
                x_anterior = x;
                switch(posicion)
                {
                    case 0:
                        if (circuito[y/10][(x + 9)/10] == 1 ||
```

```
                (y%10 != 0 && circuito[(y+7)/10][(x+9)/10] == 1))

                velocidad = 0;
            else
                x = x + 1;
            break;
        case 1:
            if (circuito[(y-2)/10][x/10] == 1 ||
                (x%10 != 0 && circuito[(y-2)/10][(x+7)/10] == 1))

                velocidad = 0;
            else
                y = y - 1;
            break;
        case 2:
            if (circuito[y/10][(x-2)/10] == 1 ||
                (y%10 != 0 && circuito[(y+7)/10][(x-2)/10] == 1))

                velocidad = 0;
            else
                x = x - 1;
            break;
        case 3:
            if (circuito[(y+9)/10][x/10] == 1 ||
                (x%10 != 0 && circuito[(y+9)/10][(x+7)/10] == 1))

                velocidad = 0;
            else
                y = y + 1;
            break;
        case 4:
            if (circuito[y/10][(x + 9)/10] == 1 ||
                (y%10 != 0 && circuito[(y+7)/10][(x+9)/10] == 1) ||
                circuito[(y-2)/10][x/10] == 1 ||
                (x%10 != 0 && circuito[(y-2)/10][(x+7)/10] == 1))

                velocidad = 0;
            else
            {
                x = x + 1;
                y = y - 1;
            }
            break;
        case 5:
            if (circuito[y/10][(x + 9)/10] == 1 ||
                (y%10 != 0 && circuito[(y+7)/10][(x+9)/10] == 1) ||
                circuito[(y+9)/10][x/10] == 1 ||
                (x%10 != 0 && circuito[(y+9)/10][(x+7)/10] == 1))

                velocidad = 0;
            else
            {
                x = x + 1;
                y = y + 1;
            }
            break;
        case 6:
            if (circuito[y/10][(x-2)/10] == 1 ||
            (y%10 != 0 && circuito[(y+7)/10][(x-2)/10] == 1) ||
            circuito[(y-2)/10][x/10] == 1 ||
            (x%10 != 0 && circuito[(y-2)/10][(x+7)/10] == 1))
```

```
                        velocidad = 0;
                    else
                    {
                        x = x - 1;
                        y = y - 1;
                    }
                    break;
                case 7:
                    if (circuito[y/10][(x-2)/10] == 1 ||
                    (y%10 != 0 && circuito[(y+7)/10][(x-2)/10] == 1) ||
                    circuito[(y+9)/10][x/10] == 1 ||
                    (x%10 != 0 && circuito[(y+9)/10][(x+7)/10] == 1))

                        velocidad = 0;
                    else
                    {
                        x = x - 1;
                        y = y + 1;
                    }
                    break;
            }
            asm("halt");
            putsprite(spr_xor,x_anterior,y_anterior,sprites[posicion]);
            putsprite(spr_xor,x,y,sprites[posicion]);
        }
    }
```

¿Y AHORA QUÉ?

Bueno, parece que al final lo hemos conseguido: tenemos un coche moviéndose por la pista, que se choca con los bordes del circuito y puede acelerar y frenar... tomando este código como base, podríamos crear nuestro fantástico juego de coches, añadir nuevas pistas, contrincantes, etc.

Sin embargo, hemos comprobado ciertas limitaciones en el código de dibujado de sprites que se suministra con z88dk, sobre todo en `juego.c`, donde se produce un ligero parpadeo del coche cuando este se mueve por la parte superior. En próximos artículos estudiaremos alguna librería que nos puede ayudar a superar el trance... pero mientras, ¡a conducir!.

Me gustaría agradecer tanto a Horace como a NoP por los comentarios realizados y la ayuda que me han prestado durante la realización de este artículo.

SIEW



- Código fuente de los ejemplos



ENTREVISTA A QUASAR SOFT

Con motivo de la celebración la nueva edición de Madrisx & Retro tuvimos la ocasión de conocer allí a los desarrolladores del que ha sido el juego estrella de este año, Vega Solaris. Fernando Saénz Pérez y Carlos García Cordero, miembros de Quasar Soft y autores del mencionado juego, se han prestado a contestar a esta interesante entrevista.

¿Cómo os conocéis y decidís empezar a desarrollar software para Spectrum?

Fernando: Es curiosa la relación entre los videojuegos y el cine en varios aspectos. Incidentalmente, estaba involucrado en la preproducción de un mediometraje (Task Computer) aficionado para el que necesitaba software. Nuestros medios eran escasos y recurrimos a programas copiados en cinta. En aquella época (año 1983) estudiaba primer curso de Ciencias Físicas y Antonio, el hermano de Carlos y que estudiaba conmigo Físicas, nos presentó para que me proporcionase estos programas. Con el tiempo y dada mi experiencia en la programación y nuestra afición por los videojuegos, decidimos crear el videojuego al que nos hubiese gustado jugar. Como a Carlos no se le daban mal los gráficos, nos repartimos de ese modo el trabajo, aunque más tarde Carlos demostró una gran competencia como programador (él es el responsable del juego de marcianitos que aparece durante la carga del programa). También es responsable de otro par de cosas: el nombre de Vega Solaris procede del original Bego (una chica conocida de Carlos), pero en Dinamic pensaron que no era adecuado y Azpiri sugirió este otro. El nombre de Quasar, y más tarde Eclipse, como grupo son también idea de Carlos.



"Todos tenemos un poco de este gusto por lo antiguo, sobre todo cuando ocurre en una época tan influyente de nuestras vidas."

Carlos: Fernando ya ha contestado casi todo. Sólo añadir que cuando nos conocimos yo ya tenía la idea del juego y algunos gráficos. En aquella época Fernando estaba pensando en hacer un juego basado en la película "Tron". Estaba con la fase tipo frontón y quería mejorar los gráficos. Al principio estuvimos ayudándonos mutuamente en nuestros respectivos proyectos (llegué a hacerle varios gráficos) y al cabo de no mucho tiempo decidimos centrarnos en Vega.

¿Dónde y cómo adquiristeis los conocimientos informáticos/gráficos necesarios para crear Vega Solaris?

Fernando: Mi afición por la informática comienza muy temprano siguiendo mi afición por la electrónica. La revista Elektor publicó un entrenador digital basado en el Rockwell 6502 que podía construirse con un sencillo circuito dotado de teclado y display de 7 segmentos, que fue el predecesor del Rockwell AIM65. Creo que costaba alrededor de 5.000 pts de entonces y mis bolsillos no estaban preparados para ello. Curiosa y coetáneamente, con la consola de juegos Videopac de Philips podía comprarse un cartucho que permitía la programación en ensamblador. No obstante, por aquel entonces no comprendía la necesidad de programar un algoritmo de multiplicación en ensamblador si la propia consola proporcionaba una calculadora. Las instrucciones, escuetas y en inglés, no ayudaron mucho. Más adelante mi padre me regaló el ZX 81, la

verdadera introducción de la informática doméstica, un lujo con un maravilloso manual de usuario que te hacía devorar páginas con miedo a que terminase.

Con esta primera época Basic desarrollo las capacidades básicas de programación (estructuras de control, bucles y rutinas) y hago mis pinitos en código máquina gracias a un libro inglés sobre programación en ensamblador del Z80. Posteriormente me regaló el Spectrum, en el que desarrollé la programación en ensamblador y pude exprimirle casi todos sus recursos. Algunos artículos de las revistas de entonces proporcionaban trucos y pistas que también me ayudaron en la programación. Como casi todos los programadores de la época, adquirí estos conocimientos de forma autodidacta. Como anécdota curiosa, en cuarto curso de Ciencias Físicas me examiné de nociones básicas de programación en ensamblador. Obtuve simplemente un compensable, es decir, un suspenso con opciones de hacer media. Mi error fue optimizar el código para tiempo y memoria al igual que estaba acostumbrado con el Spectrum. Creo que no dedicaron tiempo a analizar mi solución.

Carlos: A mí me gustaban los videojuegos y pintar. Mi padre le regaló un ZX81 a mi hermano, y acabé usándolo yo. Aprendiendo a usarlo y cómo funcionaba. Como mi padre me vio afición, me regaló un Spectrum con todos sus colorines :-). Poco a poco me fui introduciendo en el mundillo leyendo revistas y experimentando. En aquella época no había Internet y era más difícil conseguir información.



Vega Solaris en su versión final

El proceso de programar el juego fue muy artesanal. ¿De qué medios disponíais entonces para desarrollar?

Fernando: Casi la totalidad del juego lo desarrollé con el Spectrum y el casete. Por dar una idea a grandes rasgos y sin que pueda dar fe de estos números, esto significa que dediqué un 10% a desarrollo y el resto a depuración. Si en pruebas el juego se colgaba, había que reiniciar, volver a cargar el código ensamblado, los gráficos, el ensamblador y el fuente que se estaba depurando. Todo ello a 300 baudios. Es decir, al menos cinco minutos para poder recuperar la situación anterior. En ensamblador es fácil olvidar o equivocarse en una dirección cuando no se tiene todo el fuente. El proceso de reubicación de código era un infierno e intentaba evitarlo a toda costa. Si debido a la depuración alguna rutina se excedía en tamaño, intentaba acortarla usando otra lógica e instrucciones que ocupasen menos en memoria. En la última época compré un microdrive y esto agilizaba la carga del ensamblador y de los bloques de memoria preensamblados. No obstante, no eran muy fiables.

Carlos: Los medios eran un Spectrum y un casete viejo. Pasado cierto tiempo me regalaron un casete que ¡hasta tenía cuentavueltas!. Yo en aquella época era muy joven y no tenía mucho dinero.

¿Creéis que con más medios el juego se hubiera acabado antes y hubiera sido mejor? ¿O pensáis que esa falta de medios os hizo superar las dificultades y ser aún más perfeccionistas?

Fernando: Con absoluta seguridad puedo decir que un ensamblador cruzado hubiera acortado el tiempo de desarrollo drásticamente. Sin embargo, dudo que hubiésemos hecho algo cuantitativamente mejor, ya que nuestro objetivo era claro y creo que lo alcanzamos. No obstante, todo se puede mejorar, como el sonido, otro tipo de concepción de los gráficos (otra perspectiva, etc), la inteligencia artificial (IA), etc.

Carlos: Habríamos terminado mucho antes. El proceso de localizar las cosas en la cinta era lento y muchas veces fallaba la carga, con lo que había que volver a empezar. Más que hacer algo mejor, habríamos podido hacer más cosas. Habríamos terminado el juego y podríamos habernos puesto a hacer otros, con planteamientos de inicio más ambiciosos por lo menos en el apartado gráfico.

En cuanto al diseño de los gráficos, ¿usasteis algún programa para dibujarlos o recurríais al papel cuadriculado?

Fernando: Mi único papel en el diseño de los gráficos fue contactar con otro amigo de estudios de la carrera que diseñó el talismán de Vega

Solaris, tanto la punta de flecha completa como su descomposición en fragmentos, igualmente imposibles. Estaban inspirados en el trabajo de M.C. Escher e hizo un gran trabajo. Lamentablemente no recuerdo su nombre para hacerle los merecidos honores.

Carlos: Al principio usaba el Melbourne Draw, cuando el programa no daba más de sí para las cosas que yo quería, empecé a hacerme uno por mi cuenta (con GUI tipo Windows). Pero luego conseguí el Artist y me encantó, descubrí que era posible salir del programa sin borrar la pantalla. Con lo que entonces hice un programa que lanzaba el Artist y luego tomaba el control para tratar los gráficos como a mí me interesara.

Para hacer el mapa, primero me hice uno en papel “de alto nivel” para hacerme una idea de la conectividad, por dónde tenían que ir las cosas y cómo distribuir las distintas zonas (templo, jungla, grutas...). Luego las pantallas las hacía con un programa que escribí para ello.

Sorprende sobre todo la inteligencia artificial que se le ha dotado al protagonista controlado por el ordenador, ¿usasteis algún algoritmo de IA propio u os basasteis en alguno ya existente?

Fernando: El algoritmo de IA es un desarrollo personal y fue muy edificante. Por su complejidad partí de una descripción de reglas en alto nivel usando pseudocódigo (lenguaje natural) que "compilé" posteriormente a ensamblador. Realmente, después de tantos años terminado el desarrollo, cuando vi funcionar el juego de nuevo me sorprendió el nivel al que se llegó; no tanto por su funcionalidad, sino por haberla podido incluir en tan poco espacio de memoria. El objetivo, dadas las limitaciones de memoria, fue crear un oponente lo suficientemente listo como para que te complicase el juego. Como detalles puedo indicar que no había espacio ni siquiera para conservar memoria del camino recorrido salvo para unas pocas pantallas y, ni mucho menos, para incorporar un algoritmo de búsqueda de caminos en un grafo. Me limité a implementar estrategias del tipo: "Dirigirse en dirección noroeste", "Intentar alcanzar la puerta; si no es posible, retroceder por un camino diferente", "Si un obstáculo impide tu paso, intentar sortearlo, si no es posible en un margen de tiempo razonable, renunciar", "Intentar coger un objeto de interés". Apliqué un sistema de prioridades para coger y usar los objetos, los de mayor prioridad para coger eran los cristales y, para usar, los que proporcionaban mayor poder de ataque y defensa. La IA actual de los juegos no es diferente a la de entonces, la diferencia es que ahora se usan scripting, motores de inferencia y muchas reglas, pero las reglas siguen siendo fundamentalmente las mismas. Actualmente se puede modificar la IA y ver los nuevos comportamientos sin necesidad de recompilar. Entonces había que hacerlo con todos los efectos desagradables que conllevaba.

Carlos: Esto lo hizo todo Fernando, yo me limité a hablar con él las reglas de alto nivel que me comentaba por si podía aportarle alguna idea y a hacer de beta-tester.

Personalmente pude comprobar lo metódicos y ordenados que erais etiquetando, comentado todo tipo de código en papel, realizando mapas de memoria, etc. Supongo que resultaría fundamental todo esto para no perderse a la hora de depurar o verificar el funcionamiento del juego...

Fernando: Intento ser consciente de mis limitaciones, y una de ellas es mi memoria. Sabía que era fundamental documentarlo todo para atar todos los cabos en un contexto tan complejo como el desarrollo casero de productos con intención profesional. Con decenas de variables globales y rutinas con múltiples puntos de entrada y de salida, es imprescindible llevar buena cuenta de todo ello. Para crear una nueva rutina generalmente desarrollaba la idea en papel y sólo cuando tenía las cosas claras las implementaba; y con decir claro me refiero a tener escrita la rutina en papel y repasada. La depuración era tan frustrante que tenía que asegurarme de acotar el mayor número de errores.



Proceso de carga con minijuego incluido

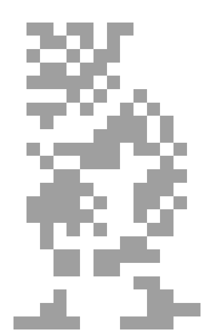
Carlos: Yo tenía en papel gran parte de las cosas que iba haciendo. Tenía dibujos de los gráficos en papel para ayudarme a hacer los mapas, aunque

con el paso del tiempo ya me los iba aprendiendo de memoria. Hice menos código que Fernando, pero con la limitación de medios que teníamos no teníamos más remedio que ser cuidadosos.

¿Teníais algún contacto con Dinamic durante el desarrollo inicial de Vega Solaris? ¿Se interesaban por el proyecto? ¿Recibisteis algún tipo de ayuda por su parte?

Fernando: Creo que el proyecto estaba suficientemente avanzado cuando le dieron el visto bueno pero yo tuve pocos contactos con ellos durante el desarrollo. Nos pareció que el proyecto sí les resultaba interesante, pero no para tirar cohetes o, al menos, es la sensación que nos transmitieron.

Carlos: Dinamic no intervino apenas nada en el desarrollo del juego, solamente en el dibujo que hizo Azpiri. Todas las ideas son nuestras para bien o para mal. Económicamente, Dinamic invirtió muy poco en nosotros. A mí me adelantaron 15.000 pesetas y creo que fue todo lo que nos dieron. En aquella época Dinamic tenía bastantes proyectos, por lo que sé funcionaban de esta manera, se gastaban poco en ellos, así no tenían mucho que perder si las cosas iban mal.



¿Os sentisteis presionados tras ver el anuncio del juego Vega Solaris en la revista Microhobby?

Fernando: El contacto con Dinamic era tan parco que ni siquiera nos avisaron de su publicación, nos cogió de improviso. Esto nos sorprendió, además considerando que no habíamos fijado fecha de fin de proyecto ni habíamos firmado ningún contrato. Obviamente sí fue una obligación moral acabar cuanto antes pero, dado que mientras estudiaba Físicas estaba además trabajando, mi tiempo era muy limitado.

Carlos: No nos consultaron nada para hacer el anuncio, con lo que fue una sorpresa. No recuerdo si nos lo dijeron con algo de antelación, supongo que sí. Nosotros trabajamos todo lo rápido que podíamos, pero hay que tener en cuenta la falta de medios así como que no era una dedicación plena. Fernando tenía que estudiar en la facultad y yo en el instituto.

¿Buscasteis más distribuidoras para vuestro juego antes o después que Dinamic? ¿Cuáles fueron sus respuestas?

Fernando: Que yo recuerde contactamos en primer lugar con Dinamic. Después hablamos con muchas otras y en general demandaban versiones para otras plataformas que para nosotros eran inabordables por la falta de experiencia y fundamentalmente por la escasez de tiempo. Anecdóticamente, algunas nos pidieron los fuentes antes de firmar. Creo que sólo a una les dejamos una demo.

Carlos: Hablamos con algunas y, en general, el juego gustó. Nos decidimos por Dinamic, ya que nos ofrecían tratos similares y Dinamic tenía más nombre. A posteriori, puede que hubiera sido mejor decidirse por una que nos apoyara más en el desarrollo.

Cuando Dinamic os propuso realizar las versiones para el resto de sistemas y no pudisteis aceptar por falta de tiempo, provocó que Dinamic descartara vuestro proyecto, ¿cuál fue vuestra reacción ante esta decisión?

Fernando: Creo que la bajada de precios del software de 2.500 a 850 pesetas fue determinante para reducir el presupuesto para el desarrollo de los juegos y nos afectó directamente. Ante las nuevas condiciones económicas vimos que no compensaba su publicación e intentamos otras distribuidoras, que resultaron vías también infructuosas. Añadiendo a esto la exigencia de conversiones para otras plataformas, el juego terminó no publicándose. La caída de precios afectó a todas las empresas y sólo Dinamic sobrevivió a aquella época.

Carlos: El trato con Dinamic era muy distante, no es que nos gustara la situación, pero tampoco nos lo tomamos muy a mal.

¿Cuánto tiempo os ha llevado finalmente el desarrollo de Vega Solaris?

Fernando: Carlos y yo nos conocimos en el año 82. No recuerdo que pasase mucho tiempo hasta que decidimos hacer el juego. Si contamos que en el 89 aparece firmada la versión definitiva son alrededor de siete años, pero si contamos que no hemos encontrado la última versión con el gráfico

de la calavera que marca el avance del tiempo y que, gracias a la ayuda de Miguel García Prada, lo hemos incluido actualmente, pues serían 23 añitos. Esto es lo que diferencia los desarrollos caseros de los profesionales.



Fernando (Izquierda) y Carlos durante la preservación del juego en la pasada Madrisx & Retro

En la pasada MadriSX & Retro se pudo recuperar gran parte de las demos, versiones beta y finales del juego, ¿qué supuso para vosotros que tras casi 20 años, vuestro juego viera al fin la luz?

Fernando: Es algo que tenía olvidado hasta tal punto que no recordaba parte de la funcionalidad del juego e incluso de los gráficos y, por tanto, me sentí incluso sorprendido. En la propia MadriSX & Retro me hicieron una pregunta similar: cómo veía el juego después de tantos años. Lo primero que contesté fue: primitivo; claro, ha llovido mucho desde entonces en el mundo de los videojuegos, aunque la pregunta iba más por estos otros derroteros.

Pero mi mayor reacción ha sido la de alegría y satisfacción de que personas como Josetxu Malanda, Juan Pablo López-Grao, José Manuel Claros, Miguel García Prada y Santiago Romero hayan hecho posible su difusión y hayan creado nuevos materiales. Dada la expectación que provocó me siento obligado a aportar todo lo que esté en mi mano proporcionar, como los mapas de memoria, los fuentes e incluso pensar en una versión e instrucciones en inglés para nuestros colegas internacionales. En verdad me siento sorprendido y halagado de que el juego haya sido tan bien acogido actualmente.

Carlos: Ha sido una gran sorpresa y estoy encantado, estoy muy halagado de que haya tanto interés. En cuanto a la versión beta (más bien una alpha, ya que era bastante primitiva) que estaba corriendo por ahí, esa versión la pasé yo a PC, no pude pasar la última debido a la carga especial que hice. Sin embargo sólo se la di a Fernando, se la enseñé a alguno que otro amigo y ya está. Como era una versión tan primitiva nunca pensamos en publicarla. No sabemos cómo llegó a distribuirse por Internet. Supongo que como tengo juegos de Spectrum en CDs, debí de prestar ese CD a alguien, ese alguien a otro alguien... y así llegó hasta un alguien que la reconoció.

Por otro lado, en este evento, pudisteis comprobar en persona cómo se encuentra actualmente el mundo de la retroinformática, ¿qué impresión os dio? ¿Conocíais de antemano algo del mundillo: emuladores, nuevo software, nuevo hardware...?

Fernando: Fue una sorpresa encontrar a tanta gente y tan bien preparada en este mundo. Aunque conocía algún emulador, no era completamente consciente de este movimiento retro hasta mi asistencia a MadriSX & Retro'2005. Creo que todos tenemos un poco de este gusto por lo antiguo, sobre todo cuando ocurre en una época tan influyente de nuestras vidas.

Carlos: Conocía algo de emuladores. Tengo afición a los emuladores aunque últimamente lo sigo menos. No obstante, nunca había estado en una feria y ha sido muy agradable encontrarse con gente con tantas ganas y tanto interés por aquellos tiempos.

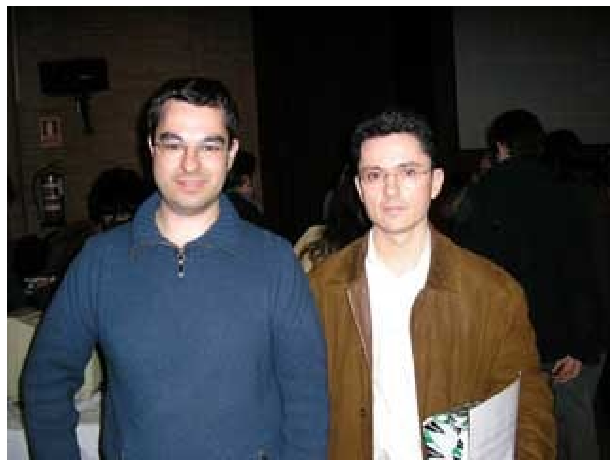
Ambos, de un modo u otro, seguís o habéis estado ligados al tema de la programación de software, ¿la experiencia con el Spectrum fue determinante para haceros seguir por este camino?

Fernando: Mi primer programa lo escribí en el Videopac de Philips (1979), mi último programa lo retoqué la semana pasada (2005). Entre estas dos fechas he programado asiduamente usando múltiples plataformas, desde ordenadores domésticos a mainframes. Aún actualmente, como profesor titular de Universidad, sigo en ello, aunque se supone que quizás no debería. Tenía relativamente claro mis objetivos formativos desde temprano. El Spectrum fue la máquina en la que desentrañé la arquitectura de un procesador y, por tanto, anticipadora de conocimientos que fundamentaría más adelante. El desarrollo de Vega Solaris sobre el Spectrum fue una tarea ardua que se completó por tres motivos fundamentales: era el juego al que más nos hubiese gustado jugar, suponía un desafío y era/soy un empecinado. Se puede decir que fue

una piedra en el camino, pero una piedra muy importante.

Carlos: Aunque no llegáramos a comercializar el juego, la experiencia fue muy positiva. Te sientes muy orgulloso cuando ves moviéndose algo en lo que has estado trabajando durante bastante tiempo y encima la gente a la que se lo enseñas le gusta.

Desde luego la experiencia no fue negativa para dejar el mundo de la programación de videojuegos. Luego he seguido en ese mundillo durante bastantes años aunque en la actualidad, por circunstancias, ya no lo esté. Trabajar con Fernando fue muy agradable y creo que conseguimos sacar adelante algo muy interesante.


Horace y Fernando posando para la posteridad

Gracias por atendernos y, especialmente, por haber mantenido guardado todo este material sobre Vega Solaris. Simplemente, si queréis añadir o decir algo a nuestros lectores, este es vuestro espacio...

Fernando: Retomando el tema del cine que mencioné al principio de la entrevista, quisiera incidir en la relación entre ambos medios, no sólo por el interés o fascinación que pueden presentarse simultáneamente por ambos. En la pasada mesa redonda "Historia del videojuego en España" que se celebró en abril de 2005 en la Universidad de Zaragora y hablando con Gonzalo Suárez (alias Gonzo, autor de Commandos y fundador de Pyro Studios, la empresa española de videojuegos con mayor impacto internacional) vi que él también tiene sus raíces en el cine. Observando las producciones recientes en videojuegos, se hace evidente que en las presentaciones se elabora material cinematográfico en el que participan elementos tales como el montaje, la narrativa, el guión, el ritmo, la banda sonora y los efectos sonoros. Esto se extiende al videojuego en sí y somos testigos de aventuras cada vez más parecidas a películas interactivas.

Mi participación más activa en el desarrollo de un videojuego de orientación profesional ha sido Vega Solaris; sin embargo, he estado y estoy relacionado con proyectos gráficos y videojuegos a lo largo de mi carrera, aunque a partir de entonces relegados a un segundo plano. En concreto he dirigido algunos proyectos de sistemas informáticos de la Ingeniería de Informática de la UCM como simuladores de vuelo de helicóptero y un videojuego en primera persona del tipo shoot'em up. Los simuladores los desarrollamos por una parte en C++ y DirectX y por otra con Simulink sobre Matlab y el paquete de realidad virtual. También participo en un proyecto del Departamento de Arquitectura de Computadores y Automática para el control automático de vuelo de helicópteros a escala, para el que hay que desarrollar el software de simulación, control y construir la mecánica de la maqueta. Colaboramos en este proyecto con el Instituto de Automática Industrial del CSIC (Centro Superior de Investigaciones Científicas), que actualmente ha desarrollado una plataforma para llevar a cabo las mediciones de parámetros físicos de la maqueta con objeto de incorporarlos en el simulador de vuelo.

Finalmente quisiera indicar mi participación en el máster en desarrollo de videojuegos de la UCM, que se imparte en colaboración con Pyro Studios. Mi papel en este máster es la dirección de un proyecto que llevan a cabo tres alumnos. El proyecto está inspirado en la película (otra relación entre videojuegos y cine) Minority Report y en el primer hito de abril los alumnos han presentado un ejecutable del juego que consiste en carreras de coches magnéticos que circulan por cualquier superficie, ya sea suelo, pared con cualquier inclinación o techo. Este ejecutable ya es jugable y causa adicción, signo de que el proyecto, cuando termine, tiene visos de ser muy interesante. Como colofón, quisiera recalcar la importancia que tiene presentar un proyecto cerrado (un videojuego que funcione) a los (pocos) empresarios de videojuegos en nuestro país si queremos trabajar en ello, como así coincidimos por ejemplo con Gonzo.

Para saber más, se puede consultar http://www.fdi.ucm.es/profesor/fernan/fsp/

Carlos: Yo no soy tan elocuente como Fernando, gracias a vosotros por vuestro interés y por ayudarnos a recuperar el Vega de las cajas de cintas viejas.

HORACE

programación ensamblador

INTRODUCCIÓN Y CONCEPTOS BÁSICOS

Todos aquellos que hayáis programado en BASIC conoceréis sin duda las limitaciones de este lenguaje de alto nivel: a cambio de su sencillez pagamos una penalización enorme en velocidad. BASIC es un lenguaje interpretado, lo que quiere decir que el "sistema operativo" (más bien el intérprete BASIC integrado en la ROM) del Spectrum tiene que leer línea a línea nuestro programa, decodificar lo que estamos diciendo en lenguaje BASIC y ejecutarlo.

Eso implica que cada vez que se ejecuta el programa, para todas y cada una de las líneas, no sólo se está ejecutando nuestro programa sino que debajo de él tenemos a la CPU del Spectrum (que no es especialmente potente) ejecutando un intérprete de BASIC que nos roba tiempo de ejecución y hace que un programa diseñado e implementado de una forma elegante se ejecute con una lentitud que no podemos salvar.

LOS LÍMITES DE BASIC

BASIC tiene una serie de trucos más o menos conocidos para acelerar su ejecución: escribir muchas instrucciones en una sóla línea BASIC, poner las rutinas que más velocidad necesitan en las primeras líneas de programa, reducir el nombre (en longitud) de las variables, etc. Pero al final llegamos a un punto en que no podemos mejorar nuestros programas en cuanto a velocidad. Sin duda, BASIC es un comienzo prácticamente obligado para programar, pero no debería ser el final. Dejando de lado que sigue siendo una herramienta muy útil para programar en el Spectrum, para muchos llega la hora de dar el siguiente paso.

Una de las primeras posibilidades que se nos plantea más allá del Intérprete BASIC del Spectrum es la utilización de un compilador de BASIC, como por ejemplo MCODER: seguimos programando en BASIC, pero lo hacemos dentro de un entorno de desarrollo (dentro del mismo Spectrum) que cuando terminamos de introducir nuestro programa, actúa como el intérprete BASIC, sólo que en lugar de ejecutar el programa lo compila y lo graba directamente en el formato que entiende el Z80. A partir de un programa en BASIC obtenemos (por ejemplo en cinta) un ejecutable que podremos cargar directamente desde el cassette. La labor de interpretación se hace igualmente, pero se hace antes, ya que en lugar de ejecutar, el resultado de la interpretación se graba en cinta. Un programa en BASIC compilado y ejecutado de este modo es muchísimo más rápido que el mismo programa ejecutado en el intérprete de BASIC del Spectrum.



Lenguaje BASIC y su intérprete

MCODER es una buena solución, y para muchos puede ser suficiente para muchas de sus creaciones. Nuestra querida DINAMIC realizó sus primeros juegos en BASIC y los compiló con MCODER: hablamos de Babaliba, Saimazoom, o la utilidad Artist. La pega es que MCODER tiene unas limitaciones que no tienen porqué ser especialmente problemáticas si las conocemos, las aceptamos, y realizamos nuestros programas teniéndolas en cuenta. Por ejemplo, no podemos utilizar vectores (creados con DIM en BASIC), y el

manejo de cadenas sufre algunos cambios de sintaxis, entre otros.

ALTERNATIVAS A BASIC

Aparte de compilar BASIC existen 3 alternativas más para programar juegos y aplicaciones que expriman al máximo nuestra máquina:

Para empezar, como primera opción, podemos realizar pequeñas rutinas en ensamblador y utilizarlas desde nuestros programas en BASIC. Posteriormente veremos todo lo necesario sobre el lenguaje ensamblador, pero como muchos de vosotros ya sabéis, se trata del lenguaje más cercano a lo que es el código binario que entiende directamente un microprocesador. El lenguaje ensamblador es de bajo nivel, es decir, está más lejos del lenguaje humano de lo que está BASIC, y a la vez está muy cerca del lenguaje que entiende el microprocesador de nuestro Spectrum.

En BASIC, una instrucción es traducida por el Intérprete BASIC a una serie más o menos larga de comandos en lenguaje máquina. Por ejemplo, 10 PRINT "HOLA", se traduce como una serie de comandos en lenguaje máquina que podrían ser algo como "para cada una de las letras de la palabra HOLA, realiza todas las operaciones necesarias para mostrar en pantalla todos los píxels que forman dichas letras, actualizando la posición del cursor y usando tal y cual color". Una instrucción BASIC equivale a muchísimas instrucciones en código máquina. Por contra, una instrucción en lenguaje ensamblador equivale a una instrucción en lenguaje máquina: hablamos directamente el lenguaje de la máquina, sólo que en vez de hacerlo con unos y ceros, lo hacemos en un lenguaje que tiene unas determinadas reglas de sintaxis y que el "programa ensamblador" se encarga de traducir a código máquina. Es por eso que programar en ensamblador es de "bajo nivel": hablamos directamente al nivel de la máquina, y por eso mismo los programas son más complicados de escribir, de leer y de mantener que un programa en BASIC, donde se habla un lenguaje más natural y que es traducido a lo que la máquina entiende.

```
; Listado 1
; Ejemplo de rutina de multiplicacion en ASM.
; El registro HL obtiene el valor de H*E .
; por David Kastrup (Z80 FAQ).
        LD    L, 0
        LD    D, L
        LD    B, 8

MULT:   ADD   HL, HL
        JR    NC, NOADD
        ADD   HL, DE
NOADD:  DJNZ  MULT
```

Así, realizamos una rutina o un conjunto de rutinas en ensamblador. Mediante un programa ensamblador, traducimos el código ASM a código que entiende directamente la máquina (código binario) y lo salvamos en cinta (o si es corto, anotamos sus valores para meterlos en DATAs) y mediante una serie de procedimientos que veremos más adelante, metemos ese código binario en memoria y lo llamamos en cualquier momento desde BASIC.

Por otro lado, nuestra segunda opción: aprender lenguaje C, y realizar programas íntegramente en C que son compilados (al igual que hace MCODER) y trasladados a código binario que ejecutará el Spectrum. Podemos ver el lenguaje C (en el Spectrum) como una manera de realizar programas bastante rápidos saltándonos las limitaciones de BASIC. No llega a ser ensamblador, y desde luego es mucho más rápido que BASIC (y que BASIC compilado). C es un lenguaje muy potente pero tal vez sea demasiado complejo para mucha gente que quiere hacer cosas muy concretas de forma que C se convierte en algo así como "matar moscas a cañonazos". Para quien ya conozca el lenguaje C y se desenvuelva bien con él, utilizar un compilador cruzado como Z88DK será sin duda una de las mejores opciones. Programando en C se puede hacer prácticamente cualquier aplicación y un gran número de juegos. Además, se puede embeber código ensamblador dentro de las rutinas en C, con lo cual se puede decir que no estamos limitados por el lenguaje C a la hora de realizar tareas que requieren un control muy preciso de la máquina. Para quien se decida por esta opción, nada mejor que Z88DK tal y como os estamos mostrando mes a mes en el curso de "Programación en C con Z88DK para Spectrum" de MagazineZX.

Finalmente, la tercera y última opción: nos hemos decidido y queremos hablarle a la máquina directamente en su lenguaje, ya que queremos controlar todo lo que realiza el microprocesador. Con BASIC compilado y con C, es el compilador quien transforma nuestros comandos en código máquina. Con Ensamblador, nosotros escribimos directamente código máquina. La máquina hará exactamente lo que le digamos, y nadie hará nada por nosotros. En este caso tenemos que programar la máquina en ensamblador (assembler en inglés, o ASM para abreviar). La diferencia de este modo con el primero que hemos comentado (integrar ASM con BASIC) es que no existe ni una sóla línea de BASIC (como mucho el cargador que lanza el programa) y realizamos todo en ensamblador.

Es importante destacar que el desarrollo de un programa en ASM requiere mucho más tiempo, un mejor diseño y muchos más conocimientos del hardware (muchísimos más) que utilizar cualquier otro lenguaje. Un programa en BASIC sencillo puede tener 1000 líneas, pero el mismo programa en ASM puede tener perfectamente 5000, 10000, o muchas más líneas. En ensamblador no tenemos a nadie que haga nada por nosotros: no existe PRINT para imprimir cosas por pantalla, si queremos imprimir texto tenemos que imprimir una a una las letras, calculando posiciones, píxeles, colores, y escribiendo en la videomemoria, nosotros mismos. Podemos apoyarnos en una serie de rutinas que hay en la ROM del Spectrum (que son las que utiliza BASIC), pero en general, para la mayoría de las tareas, lo tendremos que hacer todo nosotros.

Un ejemplo muy sencillo y devastador: en BASIC podemos multiplicar 2 números con el operador "*". En ensamblador, no existe un comando para multiplicar 2 números. No existe dicho comando porque el micro Z80 tiene definida la operación de suma (ADD), por ejemplo, pero no tiene ninguna instrucción para multiplicar. Y si queremos multiplicar 2 números, nos tendremos que hacer una rutina en ensamblador que lo haga (como la rutina que hemos visto en el apartado anterior).

Sé que estoy siendo duro y poniendo a la vista del lector un panorama desolador, pero esa es la realidad con el ensamblador: cada instrucción en ensamblador se corresponde con una instrucción de la CPU Z80. Si quieres hacer algo más complejo que lo que te permite directamente la CPU, te lo has de construir tú mismo a base de utilizar esas instrucciones. Una multiplicación se puede realizar como una serie de sumas, por ejemplo.

Visualmente, en BASIC para construir una casa te dan paredes completas, ventanas, escaleras y puertas, y combinándolos te construyes la casa. En ASM, por contra, lo que te dan es un martillo, clavos, un cincel, y madera y roca, y a partir de eso tienes que construir tú todos los elementos del programa.

Obviamente, no tendremos que escribir miles de rutinas antes de poder programar cualquier cosa: existen rutinas ya disponibles que podemos aprovechar. En Internet, en revistas Microhobby, en libros de programación de Z80, en la ROM del Spectrum (aprovechando cosas de BASIC), encontraremos rutinas listas para utilizar y que nos permitirán multiplicar, dividir, imprimir cadenas de texto, y muchas otras cosas.

POR QUÉ APRENDER ASM DE Z80

Está claro que cada lenguaje tiene su campo de aplicación, y utilizar ensamblador para hacer una herramienta interactiva para el usuario (con mucho tratamiento de textos, o de gráficos) o bien para hacer un programa basado en texto, o una pequeña base de datos o similar es poco recomendable.

Donde realmente tiene interés el ASM es en la creación de determinadas rutinas, programas o juegos orientados a exprimir el hardware de la máquina, es decir: aquellos programas orientados a escribir rápidamente gráficos en pantalla, reproducir música, o controlar el teclado con gran precisión son los candidatos ideales para escribirlos en ASM. Me estoy refiriendo a los juegos.

Ensamblador es el lenguaje ideal para programar juegos que requieran gran velocidad de ejecución. Como veremos en el futuro, dibujar en pantalla se reduce a escribir valores en memoria (en una zona concreta de la memoria). Leer del teclado se reduce a leer los valores que hay en determinados puertos de entrada/salida de la CPU, y la reproducción de música se realiza mediante escrituras en otros puertos. Para realizar esto se requiere mucha sincronización y un control total de la máquina, y esto es lo que nos ofrece ensamblador.

En MagazineZX hemos pensado y creado este curso con los siguientes objetivos en mente:

- Conocer el hardware del Spectrum, y cómo funciona internamente.
- Conocer el juego de instrucciones del Z80 que lleva el Spectrum.
- Saber realizar programas en lenguaje ASM del Z80.
- Aprender a realizar pequeñas rutinas que hagan tareas determinadas y que luego usaremos en nuestros programas en BASIC.
- Con la práctica, ser capaces de escribir un juego o programa entero en ASM.

Este pequeño curso será de introducción, pero proporcionará todos los conceptos necesarios para hacer todo esto. El resto lo aportará el tiempo que nos impliquemos y la experiencia que vayamos adoptando programando en ensamblador. No se puede escribir un juego completo en ensamblador la primera vez que uno se acerca a este lenguaje, pero sí que puede uno realizar una pequeña rutina que haga una tarea concreta en un pequeño programa BASIC. La segunda vez, en lugar de una pequeña rutina hará un conjunto de rutinas para un juego mayor, y, con la práctica, el dominio del lenguaje se puede convertir para muchos en una manera diferente o mejor de programar: directamente en ensamblador.

Queremos destacar un pequeño detalle: programar en ensamblador no es fácil. Este curso deberían seguirlo sólo aquellas personas con ciertos conocimientos sobre ordenadores o programación que se sientan preparadas para dar el paso al lenguaje ensamblador. Si tienes conocimientos de hardware, sabes cómo funciona un microprocesador, has realizado uno o más programas o juegos en BASIC u otros lenguajes o sabes lo que es binario, decimal y hexadecimal (si sabes cualquiera de esas cosas), entonces no te costará nada seguir este pequeño curso. Si, por el contrario, no has programado nunca, y todo lo que hemos hablado no te suena de nada, necesitarás mucha voluntad y consultar muchos otros textos externos (o al menos aplicarte mucho) para poder seguirnos.

Un requerimiento casi imprescindible es que el lector debe de conocer fundamentos básicos del sistema de codificación decimal, hexadecimal y binario. Como ya sabéis, nosotros expresamos los números en base decimal, pero esos mismos números se pueden expresar también en hexadecimal, o en binario. Son diferentes formas de representar el mismo número, y para distinguir unas formas de otras se colocan prefijos o sufijos que nos indican la base utilizada:

DECIMAL	HEXADECIMAL	BINARIO
64d ó 64	$40 ó 40h	%01000000
255d ó 255	$FF ó Ffh	%11111111
3d ó 3	$03 ó 03h	%00000011

Para seguir el curso es muy importante que el lector sepa distinguir unas bases de codificación de otras y que sepa (con más o menos facilidad) pasar números de una base a otra. Quien no sepa esto lo puede hacer con práctica, conforme va siguiendo el curso.

En realidad, intentaremos ser muy claros, máxime cuando no vamos a profundizar al máximo en el lenguaje: utilizaremos rutinas y ejemplos sencillos, prácticos y aplicados, y los ejecutaremos sobre emuladores de Spectrum o debuggers.

Y tras este preámbulo, podemos pasar a lo que es el curso en sí.

EL LENGUAJE ENSAMBLADOR

Como ya hemos comentado, el lenguaje ensamblador es un lenguaje de programación muy próximo a lo que es el código máquina del microprocesador Z80. En este lenguaje, cada instrucción se traduce directamente a una instrucción de código máquina, en un proceso conocido como ensamblado.

Nosotros programamos nuestras rutinas o programas en lenguaje ensamblador en un fichero de texto con extensión .asm, y con un programa ensamblador lo traducimos al código binario que entiende la CPU del Spectrum. Ese código binario puede ser ejecutado, instrucción a instrucción, por el Z80, realizando las tareas que nosotros le encomendemos en nuestro programa.

Este mes no vamos a ver la sintaxis e instrucciones disponibles en el ensamblador del microprocesador Z80 (el alma de nuestro Sinclair Spectrum): eso será algo que haremos entrega a entrega del curso. Por ahora nos debe bastar conocer que el lenguaje ensamblador es mucho más limitado en cuanto a instrucciones que BASIC, y que, a base de pequeñas piezas, debemos montar nuestro programa entero, que será sin duda mucho más rápido en cuanto a ejecución.

Como las piezas de construcción son tan pequeñas, para hacer tareas que son muy sencillas en BASIC, en ensamblador necesitaremos muchas líneas de programa, es por eso que los programas en ensamblador en general requieren más tiempo de desarrollo y se vuelven más complicados de mantener (de realizar cambios, modificaciones) y de leer conforme crecen. Debido a esto cobra especial importancia hacer un diseño en papel de los bloques del programa (y seguirlo) antes de programar una sóla línea del mismo. También se hacen especialmente importantes los comentarios que introduzcamos en nuestro código, ya que clarificarán su lectura en el futuro. El diseño es CLAVE y VITAL a la hora de programar: sólo se debe implementar lo que está diseñado previamente, y cualquier modificación de las especificaciones debe resultar en una modificación del diseño.

Así pues, resumiendo, lo que haremos a lo largo de este curso será aprender la arquitectura interna del Spectrum, su funcionamiento a nivel de CPU, y los fundamentos de su lenguaje ensamblador, con el objetivo de programar rutinas que integraremos en nuestros programas BASIC, o bien programas

completos en ensamblador que serán totalmente independientes del lenguaje BASIC.

CÓDIGO MAQUINA EN PROGRAMAS BASIC

Supongamos que sabemos ensamblador y queremos mejorar la velocidad de un programa BASIC utilizando una rutina en ASM. El lector se preguntará: "¿cómo podemos hacer esto?".

La integración de rutinas en ASM dentro de programas BASIC se realiza a grandes rasgos de la siguiente forma: escribimos nuestra rutina en ensamblador, por ejemplo una rutina que realiza un borrado de la pantalla mucho más rápidamente que realizarlo en BASIC, o una rutina de impresión de Sprites o gráficos, etc.

Una vez escrito el programa o la rutina, la ensamblamos (de la manera que sea: manualmente o mediante un programa ensamblador) y obtenemos en lugar del código ASM una serie de valores numéricos que representan los códigos de instrucción en código máquina que se corresponden con nuestro listado ASM.

CODE	CHARACTER	HEX	Z80 ASSEMBLER	-AFTER CBh	-AFTER EDh
0	|	00	nop	rlc b	
1	|	01	ld bc,NN	rlc d	
2	|- not used	02	ld (bc),a	rlc e	
3	|	03	inc bc	rlc h	
4	|	04	inc b	rlc l	
5	_|	05	dec b	rlc (hl)	
6	PRINT comma	06	ld b,N	rlc a	
7	[EDIT]	07	rlca	rlc a	
8	cursor left	08	ex af,af'	rrc b	
9	cursor right	09	add hl,bc	rrc c	
10	cursor down	0A	ld a,(bc)	rrc d	
11	cursor up	0B	dec bc	rrc e	
12	[DELETE]	0C	inc c	rrc h	
13	[ENTER]	0D	dec c	rrc l	
14	number	0E	ld c,N	rrc (hl)	
15	not used	0F	rrca	rrc a	
16	INK control	10	djnz DIS	rl b	
17	PAPER control	11	ld de,NN	rl c	
18	FLASH control	12	ld (de),a	rl d	
19	BRIGHT control	13	inc de	rl e	
20	INVERSE control	14	inc d	rl h	
21	OVER control	15	dec d	rl l	
22	AT control	16	ld d,N	rl (hl)	

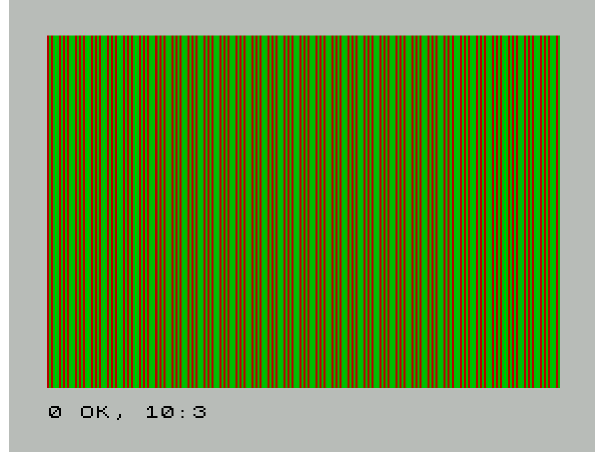
Parte de una tabla de ensamblado manual

Tras esto, nuestro programa en BASIC debe cargar esos valores en memoria (mediante instrucciones POKE) y después saltar a la dirección donde hemos POKEADO la rutina para ejecutarla.

Veamos un ejemplo de todo esto. Supongamos el siguiente programa en BASIC, que está pensado para rellenar toda la pantalla con un patrón de píxeles determinado:

```
10 FOR n=16384 TO 23295
20 POKE n, 162
30 NEXT n
```

Lo que hace este programa en BASIC es escribir un valor (el 162) directamente en la memoria de vídeo (en la próxima entrega veremos qué significa esto) y, en resumen, lo que consigue es dibujar en pantalla con unos colores determinados. Si ejecutamos el programa en BASIC veremos lo siguiente:



Salida del programa BASIC de ejemplo

Teclead y ejecutad el programa. Medid el tiempo necesario para "pintar" toda la pantalla y anotadlo. Podéis también utilizar el fichero "ejemplo1-bas.tap" que os ofrecemos ya convertido a TAP.

A continuación vamos a ver el mismo programa escrito en lenguaje ensamblador:

```
; Listado 2: Rellenado de pantalla
ORG 40000
LD HL, 16384
LD A, 162
LD (HL), A
LD DE, 16385
LD BC, 6911
LDIR
RET
```

Supongamos que ensamblamos este programa con un programa ensamblador (luego veremos cuál utilizaremos durante el curso) y generamos un TAP o TZX con él. Ejecutad el programa y calculad el tiempo (si podéis). Es en ejemplos tan sencillos como este donde podemos ver la diferencia de velocidad entre BASIC y ASM.

Para ensamblar el programa lo que hacemos es teclear el código anterior en un fichero de texto de nombre "ejemplo1.asm" en un ordenador personal. A continuación, el proceso de ensamblado y ejecución lo podemos hacer de 3 formas:

- a.- Con un programa ensamblador generamos un fichero bin (o directamente un fichero TAP) y con bin2tap generamos un TAP listo para cargar en el emulador.
- b.- Con un programa ensamblador generamos un fichero bin (que no es más que el resultado de ensamblar el código ASM y convertirlo en códigos que entiende el microprocesador Z80), los pokeamos en memoria en BASIC y saltamos a ejecutarlos.
- c.- Con un programa ensamblador generamos un fichero bin, lo convertimos a un tap sin cabecera BASIC y lo cargamos en nuestro programa BASIC con un LOAD "" CODE DIRECCION. Tras esto saltamos a la DIRECCION donde hemos cargado el código para que se ejecute.

La opción a.- es la más sencilla, y lo haremos fácilmente mediante el ensamblador que hemos elegido: PASMO. Sobre las opciones b.- y c.-, el ensamblado lo podemos hacer también con PASMO, o mediante una tabla de conversión de Instrucciones ASM a Códigos de Operación (opcodes) del Z80, ensamblando manualmente (tenemos una tabla de conversión en el mismo manual del +2, por ejemplo).

Supongamos que ensamblamos a mano el listado anterior. Ensamblar a mano consiste en escribir el programa y después traducirlo a códigos de operación consultando una tabla que nos dé el código correspondiente a cada instrucción en ensamblador. Tras el ensamblado obtendremos el siguiente código máquina (una rutina de 15 bytes de tamaño):

21, 00, 40, 3e, a2, 77, 11, 01, 40, 01, ff, 1a, ed, b0, c9

O, en base decimal:

33, 0, 64, 62, 162, 119, 17, 1, 64, 1, 255, 26, 237, 176, 201

Esta extraña cadena tiene significado para nuestro Spectrum: cuando él encuentra, por ejemplo, los bytes "62, 162", sabe que eso quiere decir "LD A, 162"; cuando encuentra el byte "201", sabe que tiene que ejecutar un "RET", y así con todas las demás instrucciones. Un detalle: si no queremos ensamblar a mano podemos ensamblar el programa con PASMO y después obtener esos números abriendo el fichero .bin resultando con un editor hexadecimal (que no de texto).

A continuación vamos a BASIC y tecleamos el siguiente programa:

```
10 CLEAR 39999
20 DATA 33, 0, 64, 62, 162, 119, 17, 1, 64, 1, 255, 26, 237, 176, 201
30 FOR n=0 TO 14
40 READ I
50 POKE (40000+n), I
60 NEXT n
```

Tras esto ejecutamos un RANDOMIZE USR 40000 y con eso ejecutamos la rutina posicionada en la dirección 40000, que justo es la rutina que hemos ensamblado a mano y pokeado mediante el programa en BASIC.

Lo que hemos hecho en el programa BASIC es:

- Con el CLEAR nos aseguramos de que tenemos libre la memoria desde 40000 para arriba (hacemos que BASIC se situe por debajo de esa memoria).
- La línea DATA contiene el código máquina de nuestra rutina.

- Con el bucle FOR hemos POKEado la rutina en memoria a partir de la dirección 40000 (desde 40000 a 40015).
- El RANDOMIZE USR 40000 salta la ejecución del Z80 a la dirección 40000, donde está nuestra rutina. Recordad que nuestra rutina acaba con un RET, que es una instrucción de retorno que finaliza la rutina y realiza una "vuelta" al BASIC.

Siguiendo este mismo procedimiento podemos generar todas las rutinas que necesitemos y ensamblarlas, obteniendo una ristra de código máquina que meteremos en DATAs y pokearemos en memoria. También podemos grabar en fichero BIN resultante en cinta (convertido a TAP) tras nuestro programa en BASIC, y realizar en él un LOAD "" CODE de forma que carguemos todo el código binario ensamblado en memoria. Podemos así realizar muchas rutinas en un mismo fichero ASM y ensamblarlas y cargarlas de una sola vez. Tras tenerlas en memoria, tan sólo necesitaremos saber la dirección de inicio de cada una de las rutinas para llamarlas con el RANDOMIZE USR DIRECCION_RUTINA correspondiente en cualquier momento de nuestro programa BASIC.

Para hacer esto, ese fichero ASM podría tener una forma como la siguiente:

```
; La rutina 1
  ORG 40000
rutina1:
  (...)
  RET

  ; La rutina 2
  ORG 41000
rutina2:
  (...)
  RET
```

También podemos ensamblarlas por separado y después pokearlas.

Hay que tener mucho cuidado a la hora de teclear los DATAs (y de ensamblar) si lo hacemos a mano, porque equivocarnos en un sólo número cambiaría totalmente el significado del programa y no haría lo que debería haber hecho el programa correctamente pokeado en memoria.

Un detalle más avanzado sobre ejecutar rutinas desde BASIC es el hecho de que podamos necesitar pasar parámetros a una rutina, o recibir un valor de retorno desde una rutina.

Pasar parámetros a una rutina significa indicarle a la rutina uno o más valores para que haga algo con ellos. Por ejemplo, si tenemos una rutina que borra la pantalla con un determinado patrón o color, podría ser interesante poder pasarle a la rutina el valor a escribir en memoria (el patrón). Esto se puede hacer de muchas formas: la más sencilla sería utilizar una posición libre de memoria para escribir el patrón, y que la rutina lea de ella. Por ejemplo, si cargamos nuestro código máquina en la dirección 40000 y consecutivas, podemos por ejemplo usar la dirección 50000 para escribir uno (o más) parámetros para las rutinas. Un ejemplo:

```
; Listado 3: Rellenado de pantalla
   ; recibiendo el patron como parametro.
   ORG 40000

   ; En vez de 162, ponemos en A lo que hay en la
   ; dirección de memoria 50000
   LD A, (50000)

   ; El resto del programa es igual:
   LD HL, 16384
   LD (HL), A
   LD DE, 16385
   LD BC, 6911
   LDIR
   RET
```

Nuestro programa en BASIC a la hora de llamar a esta rutina (una vez ensamblada y pokeada en memoria) haría:

```
POKE 50000, 162
RANDOMIZE USR 40000
```

Este código produciría la misma ejecución que el ejemplo anterior, porque como parámetro estamos pasando el valor 162, pero podríamos llamar de nuevo a la misma función en cualquier otro punto de nuestro programa pasando otro parámetro diferente a la misma, cambiando el valor de la dirección 50000 de la memoria.

En el caso de necesitar más de un parámetro, podemos usar direcciones consecutivas de memoria: en una rutina de dibujado de sprites, podemos pasar la X en la dirección 50000, la Y en la 50001, y en la 50002 y 50003 la dirección en memoria (2 bytes porque las direcciones de memoria son de 16 bits) donde tenemos el Sprite a dibujar, por ejemplo. Todo eso lo veremos con más detalle en posteriores capítulos.

Además de recibir parámetros, puede sernos interesante la posibilidad de devolver a BASIC el resultado de la ejecución de nuestro programa. Por ejemplo, supongamos que realizamos una rutina en ensamblador que hace un determinado cálculo y debe devolver, tras todo el proceso, un valor. Ese valor lo queremos asignar a una variable de nuestro programa BASIC para continuar trabajando con él.

Un ejemplo: imaginemos que realizamos una rutina que calcula el factorial de un número de una manera mucho más rapida que su equivalente en BASIC. Para devolver el valor a BASIC en nuestra rutina ASM, una vez realizados los cálculos, debemos dejarlo dentro del registro BC justo antes de hacer el RET. Una vez programada la rutina y pokeada, la llamamos mediante:

```
LET A=USR 40000
```

Con esto la variable de BASIC A contendrá la salida de nuestra rutina (concretamente, el valor del registro BC antes de ejecutar el RET). Las rutinas sólo pueden devolver un valor (el registro BC), aunque siempre podemos (dentro de nuestra rutina BASIC) escribir valores en direcciones de memoria y leerlos después con PEEK dentro de BASIC (al igual que hacemos para pasar parámetros).

CÓDIGO MÁQUINA EN MICROHOBBY

Lo que hemos visto hasta ahora es que podemos programar pequeñas rutinas y llamarlas desde programas en BASIC fácilmente. Todavía no hemos aprendido nada del lenguaje en sí mismo, pero se han asentado muchos de los conceptos necesarios para entenderlo en las próximas entregas del curso.

En realidad, muchos de nosotros hemos introducido código máquina en nuestros Spectrums sin saberlo. Muchas veces, cuando tecleábamos los listados de programa que venían en la fabulosa revista Microhobby, introducíamos código máquina y lo ejecutábamos, aunque no lo pareciera.

Algunas veces lo hacíamos en forma de DATAs, integrados en el programa BASIC que estábamos tecleando, pero otras lo hacíamos mediante el famoso Cargador Universal de Código Máquina (CUCM). Para que os hagáis una idea de qué era el CUCM de Microhobby, no era más que un programa en el cual tecleabamos los códigos binarios de rutinas ASM ensambladas previamente. Se tecleaba una larga línea de números en hexadecimal agrupados juntos (ver la siguiente figura), y cada 10 opcodes se debía introducir un número a modo de CRC que aseguraba que los 10 dígitos (20 caracteres) anteriores habían sido introducidos correctamente. Este CRC podía no ser más que la suma de todos los valores anteriores, para asegurarse de que no habíamos tecleado incorrectamente el listado.


LISTADO 2

1	3E02CD01163ACC5B5F3A	798
2	CD5B573ACE5B4F3ACF5B	1173
3	4726052E05C5E5264078	813
4	E618846778E6070F0F0F	891
5	816F06080E00C5010001	467
6	1A77ED4A13C110F4E1C1	1346
7	0C2520D926053ACE5B4F	775
8	042D20CF3AD05B5F3AD1	1007
9	5B573AD25B6F3AD35B67	1111
10	0E0506057E1223130D20	273
11	F90E053E1B835F3E008A	783
12	570520ECC90000000000	561

DUMP: 53.000
N.° BYTES: 115

Ejemplo de listado para el CUCM de MH

Al acabar la introducción en todo el listado en el CUCM, se nos daba la opción de grabarlo. Al grabarlo indicábamos el tamaño de la rutina en bytes y la dirección donde la ibamos a alojar en memoria (en el ejemplo de la captura, la rutina se alojaría en la dirección 53000 y tenía 115 bytes de tamaño). El CUCM todo lo que hacía era un simple:

```
SAVE "" CODE 53000, 115
```

Esto grababa el bloque de código máquina en cinta (justo tras nuestro programa en BASIC), de forma que el juego en algún momento cargaba esta rutina con LOAD "" CODE, y podía utilizarla mediante un RANDOMIZE USR 53000.

PASMO: ENSAMBLADOR CRUZADO

El lector se preguntará: "Y si no quiero ensamblar a mano, ¿cómo vamos a ensamblar los listados que veamos a lo largo del curso, o los que yo realice para ir practicando o para que sean mis propias rutinas o programas?". Sencillo: lo haremos con un ensamblador cruzado, es decir, un programa que nos permitirá programar en un PC (utilizando nuestro editor de textos habitual), y después ensamblar ese fichero .asm que hemos realizado, obteniendo un fichero .BIN (o directamente un .TAP) para utilizarlo en nuestros programas.

Los programadores "originales" en la época del Spectrum tenían que utilizar MONS y GENS para todo el proceso de desarrollo. Estos programas (que corren sobre el Spectrum) implicaban teclear los programas en el teclado del Spectrum, grabarlos en cinta, ensamblar y grabar el resultado en cinta, etc. Actualmente es mucho más comodo usar programas como los que usaremos en nuestro curso.

Por ejemplo, PASMO. Pasmo es un ensamblador cruzado, opensource y multiplataforma. Con Pasmo podremos programar en nuestro PC, grabar un fichero ASM y ensamblarlo cómodamente, sin cintas de por medio. Tras todo el proceso de desarrollo, podremos llevar el programa resultante a una cinta (o disco) y ejecutarlo por lo tanto en un Spectrum real, pero lo que es el proceso de desarrollo se realiza en un PC, con toda la comodidad que eso conlleva.

Pasmo en su versión para Windows/DOS es un simple ejecutable (pasmo.exe) acompañado de ficheros README de información. Podemos mover el fichero pasmo.exe a cualquier directorio que esté en el PATH o directamente ensamblar programas (siempre desde la línea de comandos o CMD, no directamente mediante "doble click" al ejecutable) en el directorio en el que lo tengamos copiado.

La versión para Linux viene en formato código fuente (y se compila con un simple make) y su binario "pasmo" lo podemos copiar, por ejemplo, en /usr/local/bin.

Iremos viendo el uso de pasmo conforme lo vayamos utilizando, pero a título de ejemplo, veamos cómo se compilaría el programa del Listado 2 visto anteriormente. Primero tecleamos el programa en un fichero de texto y ejecutamos pasmo:

```
pasmo ejemplo1.asm ejemplo1.bin
```

Con esto obtendremos un bin que es el resultado del ensamblado. Es código máquina directo que podremos utilizar en los DATAs de nuestro programa en BASIC. Podemos obtener el código máquina mediante un editor hexadecimal o alguna utilidad como "hexdump" de Linux:

```
$ hexdump -C ejemplo1.bin
 00000000  21 00 40 3e a2 77 11 01 40 01 ff 1a ed b0 c9
```

Ahí tenemos los datos listos para convertirlos a decimal y pasarlos a sentencias DATA. Si el código es largo y no queremos teclear en DATAs la rutina, podemos convertir el BIN en un fichero TAP ensamblando el programa mediante:

```
pasmo --tap ejemplo1.asm ejemplo1.tap
```

Este fichero tap contendrá ahora un tap con el código binario compilado tal y como si lo hubieras introducido en memoria y grabado con SAVE "" CODE, para ser cargado posteriormente en nuestro programa BASIC con LOAD "" CODE.

Los programas resultantes pueden después cargarse en cualquier emulador para comprobarlos (como Aspectrum, FUSE). De este modo el ciclo de desarrollo será:

- Programar en nuestro editor de textos favorito.
- Ensamblar el programa .asm con pasmo.
- Cargar ese código máquina en memoria, bin con DATA/POKE o bien cargando un tap con LOAD "" CODE.
- Realizar nuestro programa BASIC de forma que utilice las nuevas rutinas, o bien directamente programar en ensamblador.

Este último paso es importante: si estamos realizando un programa completo en ensamblador (sin ninguna parte en BASIC), bastará con compilar el programa mediante "pasmo --tapbas fichero.asm fichero.tap". La opción --bastap añade una cabecera BASIC que carga el bloque código máquina listo para el RANDOMIZE USR.

Si, por contra, estamos haciendo rutinas para un programa en BASIC, entonces bastará con generar un BIN o un TAP y grabarlo tras nuestro programa BASIC. Para eso, escribimos las rutinas en un fichero .ASM y las compilamos con "pasmo --tap fichero.asm fichero.tap". Después, escribimos nuestro programa en BASIC (con bas2tap o en el emulador). Tras esto tenemos que crear un TAP o un TZX que contenga primero el bloque BASIC y después el bloque código máquina (y en el bloque BASIC colocaremos el LOAD "" CODE DIRECCION, TAMANYO_BLOQUE_CM necesario para cargarlo). Esto, sin necesidad de utilizar emuladores de por medio, sería tan sencillo como:

```
Linux:    cat programa_basic.tap bloque_cm.tap > programa_completo.tap
Windows:  copy /b programa_basic.tap +bloque_cm.tap programa_completo.tap
```

EN RESUMEN

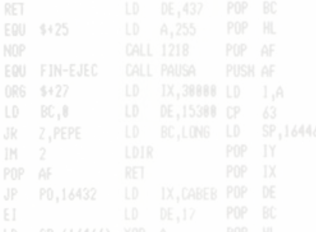
En esta entrega hemos definido las bases del curso de ASM de Z80, comenzando por las limitaciones de BASIC y la necesidad de conocer un lenguaje más potente y rápido. Hemos visto qué aspecto tiene el código en ensamblador (aunque todavía no conozcamos la sintaxis) y, muy importante, hemos visto cómo se integra este código en ensamblador dentro de programas en BASIC. Por último, hemos conocido una utilidad que nos permitirá, a lo largo del curso, ensamblar todos los programas que realicemos (y probarlos en un emulador, integrado en nuestros programas BASIC).

A lo largo de los siguientes artículos de este curso aprenderemos lo suficiente para realizar nuestras propias rutinas, gracias a los conceptos y conocimientos teóricos explicados hoy (y en la siguiente entrega).

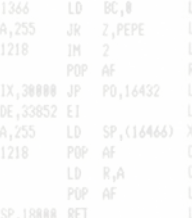
SROMERO

- Ejemplo de rellenado de pantalla en BASIC (fuente)
- Ejemplo de rellenado de pantalla en BASIC (tap)
- Ejemplo de rellenado de pantalla en ASM (fuente)
- Ejemplo de rellenado de pantalla en ASM (tap)
- Código binario del ejemplo ejemplo1.asm
- Capítulo 26 del manual de +3
- Pasmo
- Aspectrum
- Hexedit

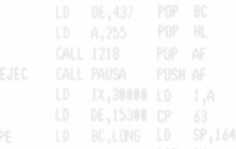


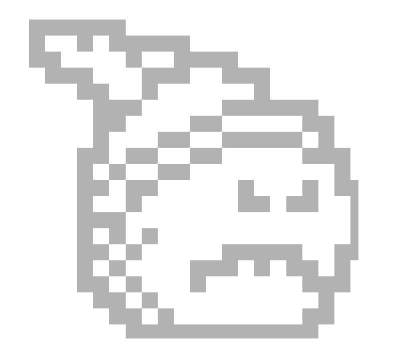












opinión

DISTRIBUCIÓN DENEGADA

En pleno mes de febrero me llega la noticia, a través de los correos de actualización de WOS, de que otra empresa que desarrollaba software para Spectrum ha denegado su ditribución, uniéndose así a los Capcom, Ultimate o Codemasters de turno.

Paul McKenna, propietario de los derechos de Thor Computer Software y Odin Computer Graphics Ltd., nos niega el "placer" de disfrutar de sus juegos, entre los que se encuentran títulos muy conocidos por todos como Nodes y Arc of Yesod o Heartland, hasta un total de 17 títulos.

¿Qué puede impulsar a una empresa a denegar la distribución de sus creaciones veinte años después? La respuesta es muy simple: DINERO. Con el auge de los teléfonos móviles y las consolas portátiles, que nos permiten jugar a versiones mejoradas pero muy similares de estos programas, los dueños de esos derechos ven un filón del cual sacar tajada a un trabajo realizado hace mucho tiempo. En muchos casos con una simple conversión; en otros, aprovechando el tirón mediático que los personajes protagonistas de los mismos tienen entre una generación que creció con ellos y a los que pueden hacer resurgir la nostalgia sin mucha dificultad y, lo que es más importante, tienen poder adquisitivo para invertir en sus productos llamados por la vena *friki* que a todos los integrantes de esa hornada nos corre por el cuerpo.

Por supuesto estas empresas son libres de hacer con las creaciones de las que poseen los derechos lo que les plazca, y no nos queda más remedio que aceptarlo aunque no estemos de acuerdo con la decisión.

Como muchos ya sabéis, quien esto escribe mantiene una página con este tipo de juegos, pero en los últimos tiempos me planteo si realmente merece la pena. Me explico: si a las personas que dieron la vida a nuestros héroes de la infancia no les interesa que sigamos disfrutando con ellos, que los añoremos y que les sigamos agradeciendo esos maravillosos momentos que pasamos en nuestra infancia, a mi no me interesa mantener vivo el recuerdo, arriesgándome a padecer problemas legales.

Y, al margen de estos problemas, que habría que ver si llegarían a darse, tenemos cientos de alternativas para pasar buenos ratos con juegos del mismo estilo que, en muchos casos, sobrepasan en calidad y adicción a aquéllos.

Ultimate nos prohíbe jugar con Sabreman al Knight Lore, pues bueno, tenemos a Head y Heels, a Batman, a Marlow esclareciendo misterios en Movie o a nuestro rechoncho protagonista de Fairlight sin dejar de lado la fantástica Abadía y su clima absorbente. Sin querer restar méritos a una empresa pionera, que lanzó el primer Filmation, mientras ellos se quedaron atascados y no hacían más que repetir el mismo juego con diferentes gráficos, los Jon Ritman, los Paco Menéndez y tantos otros avanzaban un paso por delante, llevando un poco más lejos el límite de nuestro querido Spectrum. Y no olvidemos a Sandy White y el Ant Attack, más conocido por Hormigas en estos lares o el Zombie Zombie, lanzados con bastante antelación.

¿Que Capcom no nos deja masacrar soldaditos con su Bionic Comando?, cargamos el Green Beret, o al Army Moves, o el Astro Marine Corps, o el Game Over, o cualquier título de un tipo de juegos de los que podemos estar orgullosos, ya que en España teníamos unas empresas a las que se les daba muy bien este estilo.

Y si el cabezón de Dizzy está cogiendo polvo en el fondo de un cajón, acompañado de Charlie que ya no pasea por Yesod, tenemos el Starquake, la saga Wally, nuestro gusanito pasando frío en Frost Byte y una larga e interminable lista de juegos con la que podía seguir extendiéndome durante horas.

Curiosamente estas empresas que tanto ruido hacen ahora, son las que se quedaron estancadas en su época, lanzando novedades muy llamativas en un principio y en alguno de los casos, pero repitiéndose constantemente hasta el aburrimiento después y entrando en un declive pronunciado.

Solventado el problema más serio, que puede ser el de la preservación del software y todos sus añadidos, cosa conseguida en mayor o menor grado en proyectos como WOS, TZXVAULT o SPA2, el que los podamos tener disponibles creo que, por lo menos en mi caso, es algo irrelevante.

Señores de Codemasters, Ultimate/Rare/Microsoft, Odin/Thor, Capcom que ustedes lo denieguen bien.

MIGUEL


ZX
magazine